MASTER EN SCIENCES INFORMATIQUES

Restructuration de programme en milieu paginé

Gaspard, Guy

Award date:
1976

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE DAME DE LA PAIX
INSTITUT D'INFORMATIQUE

Année académique 1975 - 1976

RESTRUCTURATION DE PROGRAMMES EN MILIEU - PAGINE

Guy GASPARD

Mémoire présenté en vue de l'obtention du grade de Licencié et Maître en Informatique

Je remercie les personnes m'ayant aidé à l'élaboration de ce mémoire.
Ma profonde reconnaissance va à Monsieur le Professeur J. RAMAEKERS dont l'aide constante, les conseils et les critiques constructives ont permis la réalisation de ce travail.
Je tiens également à remercier Monsieur G. VANGHELUWE pour l'attention qu'il a prêtée à mes travaux.
Je remercie enfin mon épouse pour le soin et la diligence qu'elle a apporté à son impression.

TABLE DES MATIERES

CHAPITRE I INTRODUCTION

1.1. La mémoire virtuelle
1.2. La restructuration

CHAPITRE II : AFFINITE

2.1. Introduction
2.2. Concept d'affinité
2.3. Evaluation théorique de l'affinité
  2.3.1. Evaluation théorique de l'affinité basée sur l'emplacement des références les unes par rapport aux autres dans la chaîne.
  2.3.2. Evaluation théorique de l'affinité basée sur le temps.
2.4. Matrice d'affinité
2.5. Méthodes d'évaluation pratique de l'affinité
  2.5.1. Introduction
  2.5.2. Calcul de l'affinité basé sur des mesures statiques
  2.5.3. Calcul dynamique de l'affinité
    2.5.3.1. Calcul dynamique de l'affinité basé sur l'emplacement des références les unes par rapport aux autres dans la chaîne.
    2.5.3.2. Calcul dynamique de l'affinité basé sur le temps.
      2.5.3.2.1. Extension des méthodes précédentes
      2.5.3.2.2. Utilisation des working-sets
      2.5.3.2.3. Méthode des working-sets critiques.
    2.5.3.3. Calcul dynamique de l'affinité par échantillonnage de la chaîne de référence.

CHAPITRE III : RESTRUCTURATION

3.1. Introduction
3.2. Algorithmes tenant compte de la pagination.de la mémoire
  3.2.1. Algorithme de RYDER
  3.2.2. Algorithme de HATFIELD et GERALD
  3.2.3. Algorithme de MASUDA
  3.2.4. Densification
3.3. Algorithmes indépendants de la pagination.

CHAPITRE IV : OPTIMISATION DES ALGORITHMES

4.1. Introduction
4.2. Recherche de l'élément de valeur maximale dans la matrice d'affinité.
4.3. Optimisation de l'accès à la table des identificateurs de blocs

CHAPITRE V : RESTRUCTURATION AUTOMATIQUE

5.1. Introduction
5.2. Importance de l'éditeur de liens
5.3. Le triangle d'auto-adaptation
5.4. Fonctionnement de l'éditeur de liens,
Possibilité d'y inclure des modules de restructuration.

CHAPITRE VI : TEST DES NOUVELLES IMPLANTATIONS

6.1. Introduction
6.2. Description de notre simulateur

CHAPITRE VII : RESULTATS DES TESTS DE QUELQUES METHODES DE CALCUL DE L'AFFINITE ET DE RESTRUCTURATION

7.1. Introduction
7.2. Résultats

CHAPITRE VIII : CONCLUSIONS

CHAPITRE 1 : INTRODUCTION

1.1. La mémoire virtuelle
1.2. La restructuration

1.1. La mémoire virtuelle.

Dans les systèmes à mémoire virtuelle, tout spécialement s'ils sont multiprogrammés les programmes en cours d'exécution ne se trouvent pas entièrement en mémoire principale. Ces programmes devront donc être scindés en portions qui seront chargées successivement en mémoire lors de l'exécution. Pour ce qui nous occupe, ces portions sont de taille fixe; ce sont les "pages". La mémoire physique est alors scindée en "cadres" de taille correspondante.
La page constitue l'unité d'allocation de mémoire et de transfert d'information entre la mémoire principale et les mémoires périphériques.

Un algorithme de gestion de mémoire détermine le nombre de cadres alloués à chaque programme ainsi que les pages d'informations qui y seront mémorisées à chaque instant, en vue de maximiser la performance du système.

De nombreux indices permettent d'évaluer la performance d'un système. Nous avons choisi de l'évaluer grâce au taux de défauts de pages ou encore au nombre de défauts de page (nombre de pages ne se trouvant pas en mémoire principale quand le processeur en a besoin) générés par un programme pendant son exécution.

Le but d'un bon algorithme de gestion de mémoire est de prédire quelles pages seront référencées dans un futur proche et de charger ces pages de façon à ce que le processeur les trouve quand il en a besoin.

L'algorithme optimal est celui qui connait toute la chaîne des références futures, il n'est donc pas réalisable.
Les recherches entreprises sur les algorithmes réalisables ont montré qu'il existe entre eux peu de différences du point de vue performance et qu'ils ne sont pas trop éloignés de l'algorithme optimal.
En dépit de cela, il s'est avéré que les transferts de pages dans un système pouvaient devenir trop nombreux et provoquer l'écroulement du système (trashing), ou du moins, une nette déterioration du temps de réponse.

1.2. La restructuration.

Il est donc nécessaire d'adapter les programmes au milieu paginé. COMEAU /12/ pense que c'est au programmeur d'avoir connaissance du milieu dans lequel sera traité son programme. Cette solution est loin d'être satisfaisante. Des recherches sur l'adaptation des programmes au milieu paginé se sont donc développées Cette adaptation porte le nom de restructuration.

Un programme étant composé d'un ensemble de blocs d'informations, une restructuration consiste à déterminer une implantation des blocs dans l'espace virtuel en fonction de mesures faites préalablement sur le programme.
On s'efforcera pour ce faire d'exploiter la "localité" des programmes en rendant contigues spatialement les parties qui le sont temporellement, c'est à dire qui sont référencées endéans des courts intervalles de temps.

Le concept de localité est défini par Tsichritzis/29/ par le fait que pendant n'importe quel intervalle de temps d'exécution un processus référence plus certains de ses constituants que d'autres, l'ensemble des constituants favorisés peut être différent pour des intervalles de temps différents.
Cette propriété est due à plusieurs facteurs :

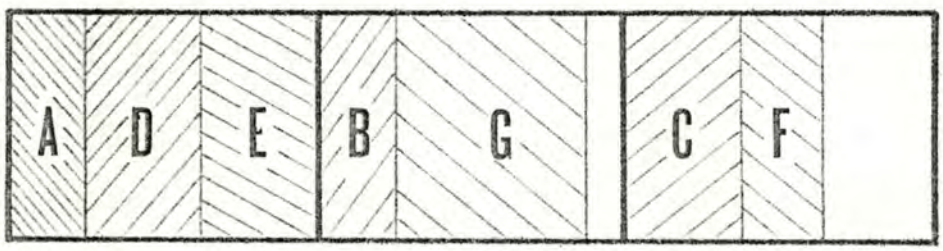
- l'organisation séquentielle des instructions :
  lorque l'on exécute une instruction, il est très probable que la suivante soit en séquence dans la même page;
- la modularité
  un programme est presque toujours découpé en modules ayant une fonction spécifique.
- la structure des informations :
  les informations référencées à un moment donné sont souvent groupées en raison des liens logiques existant entre elles (c'est le cas des données locales à une procédure)
- certains mécanismes de programmation :
  par exemple : les boucles favorisent pendant leur exécution la ou les pages qui les contiennent

L'exemple qui suit montre que la restructuration d'un programme peut effectivement diminuer le nombre de défauts de page générés lors de son exécution.

Soit un programme constitué de 7 blocs A, B, C, D, E, F, G.
Le passage de contrôle d'un bloc à l'autre lors de l'exécution se faisant selon la séquence :

A B C E C B C E C F C B D B D G D B A

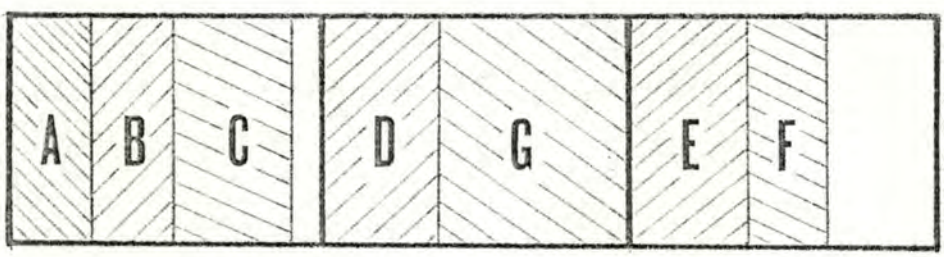
Si les blocs sont implantés comme suit :



Page 1. Page 2. Page 3.

Le programme génère huit défauts de page en supposant qu'il dispose de 2 cadres mémoire, cette mémoire étant gérée par un algorithme de type LRU.

Par contre, en gardant les mêmes hypothèses mais si les blocs sont implantés de la façon qui suit :



le nombre de défauts de page générés n'est plus que de 3. Il est possible de démontrer sur ce même exemple que la restructuration réduit la taille des working-sets du programme :.

On s'attendra donc à ce que la restructuration des programmes produise différents effets bénéfiques :

- d'un point de vue global, les working-sets des programmes restructurés étant de taille inférieure à la taille des working-sets des programmes non restructurés, un plus grand nombre de ceux ci peuvent occuper simultanément la mémoire principale.
- du point de vue de chaque programme, une diminution du taux de défauts de page réduit le temps de réponse et le temps d'exécution (elapsed time).

Certaines conditions doivent cependant être nécessairement vérifiées : il importe que le comportement dynamique du programme que l'on souhaite restructurer ne varie pas trop en fonction de ses données.
En effet, il semble que cette propriété soit vérifiée pour des programmes de taille suffisament grande. HATFIELD et GERALD /20/ ont montré la stabilité de comportement de plusieurs programmes vis à vis de leurs données. Cette stabilité a également été vérifiée à l'IRIA /25/pour le compilateur PL/1 conversationnel.

La restructuration d'un programme telle que nous l'avons envisagée comporte plusieurs étapes :

Etape 1. : le programme est divisé en blocs relogeables. Ces blocs ont une taille moyenne très inférieure à la taille d'une page.

Etape 2. : l'affinité entre les différents blocs est estimée ou calculée.

Etape 3. : un algorithme de restructuration trouve à partir de ces affinités le meilleur arrangement des blocs dans les pages.

Etape 4. : les blocs sont réarrangés dans l'espace virtuel en fonction des résultats de l'étape précédente.

CHAPITRE II : AFFINITE

2.1. Introduction
2.2. Concept d'affinité
2.3. Evaluation théorique de l'affinité
2.4. Matrice d'affinité
2.5. Méthode d'évaluation pratique de l'affinité.

2.1. Introduction

Ce chapitre débute par une définition du concept d'affinité; il se poursuit par la mise en évidence de différentes méthodes de calcul théoriques et pratiques de l'affinité existant entre les différents blocs d'un programme. La notion de bloc, unité de découpage d'un programme, étant ici simplement liée à celle d'un ensemble arbitraire de données ou d'instructions.

Nous considérerons qu'un processus peut-être représenté par une chaîne de références. Cette chaîne est constituée de toutes les références aux blocs du programme, références générées par le processeur lors de l'exécution du programme.
Si nous représentons toute référence au bloc X par la notation r_x, la chaîne prendra par exemple l'aspect suivant :

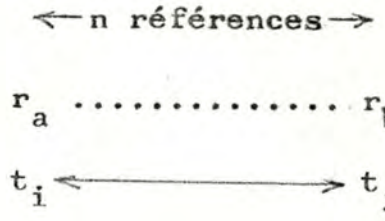
$r_a\ r_a\ r_a\ r_a\ r_b\ r_b\ r_b\ r_c\ r_b\ r_b\ r_b\ r_c\ r_b\ r_b\ r_a\ r_a\ r_a\ r_a$

dans le cas d'un programme constitué de trois blocs A, B et C.
Nous ferons l'hypothèse qu'il est possible de constituer en parallèle avec la chaîne des références, une chaîne des temps constituée des temps CPU relevés lors de chaque référence.

2.2. Concept d'affinité.

Lors d'une première approche, nous définirons la contribution à l'affinité entre deux blocs, au moyen d'une fonction décroissante du temps écoulé entre les références à ces blocs, et ce quel que soit le mécanisme de mesure de cet intervalle de temps :

Soit une chaîne contenant les références r_a et r_b espacées de n références ou d'un temps $t = t_j - t_i$



$aff(r_a, r_b) = f(n)$ | $aff(r_a, r_b) = f(t_j - t_i)$

$f(n) \leq f(n-1)$ | $t = t_j - t_i$

$0 \leq n + \leq \infty$ | $f(t) \leq f(t - dt)$
$dt \geq 0$
$0 \leq t \leq \infty$

Nous tiendrons compte de plus des conditions aux limites qui suivent :

1) $\lim_{n \to \infty} f(n) = 0$ | $\lim_{t \to \infty} f(t) = 0$

ce qui implique que des références très éloignées les une des autres ne contribuent pas à l'établissement de l'affinité entre les blocs auxquels elles réfèrent.

2) $\lim_{n \to 0} f(n) = f_n(0)$ | $\lim_{t \to 0} f(t) = f_t(0)$

Par la suite, nous verrons que les méthodes de calcul de l'affinité dues à FERRARI sont basées sur d'autres notions : entre autre, l'appartenance des références à certaines sous-chaînes de la chaîne de référence initiale.

L'affinité entre deux blocs A et B (aff(a, b)) d'un programme sera définie par la somme des contributions à l'affinité (aff(r_a, r_b)) pour ces mêmes blocs.

$$aff(a, b) = \sum_{\text{chaîne}} aff(r_a, r_b)$$

L'affinité entre deux blocs quelconques d'un programme peut donc être assimilée à la tendance qu'ont les références à ces deux blocs d'apparaître de façon fréquemment conjointe dans la chaîne de références.
Une définition similaire pourrait être énoncée pour un groupe quelconque de n ($n \geq 2$) blocs.

Différents éléments contribuent donc à l'établissement d'une affinité entre deux blocs d'un programme. Nous pouvons les classer en deux catégories :

1. L'emplacement des références les une par rapport aux autres dans la chaîne.
   - l'apparition consécutive des références dans la chaîne
   - la proximité des références dans la chaîne, comme c'est par exemple le cas pour les blocs A et B dans la chaîne : $r_a\ r_c\ r_b$

2. Le temps écoulé entre les références aux deux blocs.

2.3. Evaluation théorique de l'affinité.

Cette évaluation théorique repose sur ce qui a été mis en évidence à propos du concept d'affinité.

2.3.1. Evaluation théorique de l'affinité basée sur l'emplacement des références les unes par rapport aux autres dans la chaîne.

1. Dans le cas de la succession directe, si chacune des contributions à l'affinité entre deux blocs vaut 1 ($f_n(0) = 1$), l'affinité entre deux blocs quelconques sera égale au nombre de transitions relevées entre ces deux blocs dans la chaîne de références.

2. Le choix de la fonction f(n) qui lie la contribution à l'affinité due à des références espacées de n autres références à ce nombre constitue la principale difficulté de cette méthode.
   De façon théorique, nous pourrions par exemple faire le choix de l'une ou l'autre des fonctions suivantes :

$$f(n) = \frac{1}{1+n} \qquad \begin{array}{ll} f(n) = k(C - n) & n \leq C \\ f(n) = 0 & n > C \end{array}$$

Ces fonctions répondent aux critères et conditions aux limites que nous nous sommes imposés :

$$\lim_{n \to \infty} \frac{1}{1+n} = 0 \qquad \lim_{n \to \infty} k(C - n) = 0$$

$$\lim_{n \to 0} \frac{1}{1+n} = 1 \qquad \lim_{n \to 0} k(C - n) = kC$$

avec $kC = 1$

et $f(n+1) \leq f(n)$ car

$$\frac{1}{n+1+1} \leq \frac{1}{n+1} \qquad k(C-n-1) \leq k(C-n)$$

$$n+1 \leq n+2 \qquad C-n-1 \leq C-n$$
$$-1 \leq 0$$

2.3.2. Evaluation théorique de l'affinité basée sur le temps.

Le problème reste identique si l'on tient compte cette fois du temps écoulé entre les références.
Nous introduirons dans ce cas une fonction de ce temps par exemple :

$$f(t) = \frac{1}{1+t} \qquad f(t) = k(C-t)$$

La vérification des conditions aux limites reste identique à celle faite précédemment. Il reste à montrer que :

$$f_t(t+dt) \leq f_t(t) \qquad dt \geq 0$$

Ce qui est évident dans les deux cas.

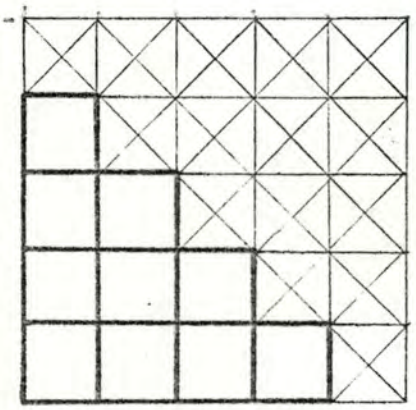
2.4. Matrice d'affinité.

Un programme peut être représenté pour ce qui nous occupe par son graphe de relation G(X, U) défini comme suit :

X = ensemble des noeuds représentant les différents blocs constituant le programme.

U = famille des arcs ou des arêtes représentant les appels possibles entre les différents blocs ainsi que les références à des données.

Nous considérerons que ce graphe est non orienté : notre but étant de rechercher quels sont les blocs qui vont être placés ensemble dans une même page ou dans un même groupe de pages, il ne nous est pas utile de savoir par exemple de deux blocs quel est le bloc appelant et le bloc appelé.
Les arêtes sont éventuellement pondérées par l'affinité existant entre les blocs qu'elles relient.
La représentation matricielle de ces affinités porte le nom de matrice d'affinité.
Cette matrice est symétrique, nous ne la mémoriserons donc jamais dans son entièreté. De plus, les éléments de la diagonale principale seront négligés.



2.5. Méthodes d'évaluation pratiques de l'affinité

2.5.1. Introduction
2.5.2. Calcul de l'affinité basé sur des mesures statiques
2.5.3. Calcul de l'affinité basé sur des mesures dynamiques

2.5.1. Introduction.

Les méthodes pratiques de calcul de l'affinité se scindent en deux groupes en fonction :

1) des mesures statiques qui peuvent être réalisées avant toute exécution du programme par examen du code source.
2) des mesures dynamiques basées sur une chaîne de références obtenue pendant l'exécution du programme. La chaîne de références décrite précédemment étant difficilement utilisable par suite des coûts élevés de son extraction et de sa manipulation, est souvent remplacée par une chaîne réduite lors des calculs d'affinité. Cette dernière ne comporte par exemple que les passages de contrôle entre les différents blocs constituant le programme. La notion de temps y est éventuellement incluse par l'indication du temps CPU relevé lors de chaque passage de contrôle du processeur.
   Cette chaîne prend par exemple l'aspect suivant :

$$r_a\ r_b\ r_c\ r_b\ r_d\ r_b\ r_a$$

   alors que pour la même exécution du programme la chaîne complète aurait pu prendre la forme :

$$r_a..r_a\ r_b.r_f..r_b\ r_c..r_c\ r_b..r_b\ r_d..r_d\ r_b..r_b\ r_a$$

où r_f représente une référence à un bloc de données, référence qui n'apparaît plus dans la chaîne réduite. D'autres possibilités de réduction de coûts existent, par exemple l'échantillonnage des chaînes de références.

Les chaîne utilisées pratiquement constituent des sous-ensembles de la chaîne théorique.

2.5.2. Calcul statique de l'affinité.

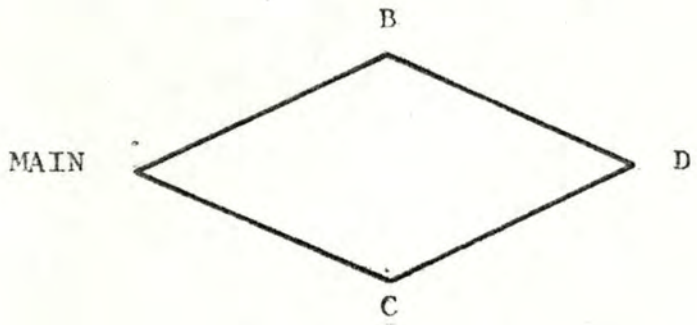
Par un simple examen du code source du programme dont on souhaite améliorer le comportement, il est possible de repérer les liens existants entre les différents blocs qui le constituent. Ce niveau correspond à un relevé des appels inter-blocs du programme.
Nous pouvons à ce stade tracer un graphe des relations du programme.

Il est intéressant de remarquer ici que ce travail peut être réalisé automatiquement par un module d'analyse du programme inclus à l'éditeur de liens. Ce module, à la condition que la notion de bloc soit identique à celle de module objet, dispose de toutes les informations qui lui sont nécessaires dans les tables du relieur.

Exemple : soit le programme FORTRAN

```
PROGRAM MAIN          SUBROUTINE B          SUBROUTINE D
.....                 .....                 .....
CALL B                CALL D                .....
.....                 .....                 RETURN
CALL C                RETURN
.....
END                   SUBROUTINE C
                      .....
                      CALL D
                      .....
                      RETURN
```

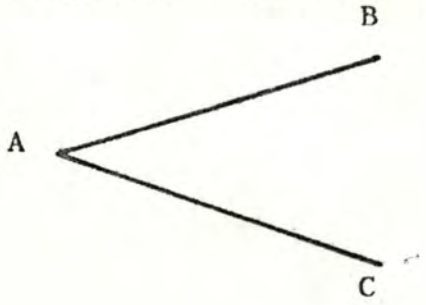
Son graphe des relations se présente comme suit :



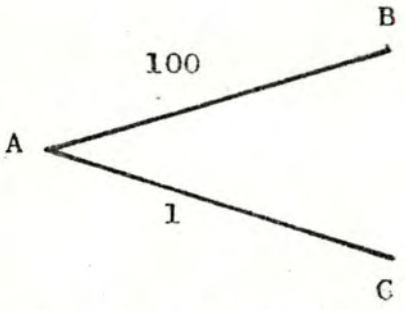
Il est possible de poursuivre la démarche entreprise en pondérant les arêtes du graphe en fonction de la probabilité qu'elles ont d'être parcourues lors des exécutions ultérieures, et de la fréquence de ces parcours.
Notre expérience personnelle aux Facultés confirme l'expérience de COMEAU et montre que ces estimations sont rarement correctes,/12/ l'exemple qui suit montre la difficulté d'évaluer la fréquence d'appel d'un bloc par un autre.

```
SUBROUTINE A
.....
DO 100 I = 1, 100
CALL B
100 CONTINUE
.....
CALL C
.....
RETURN
```

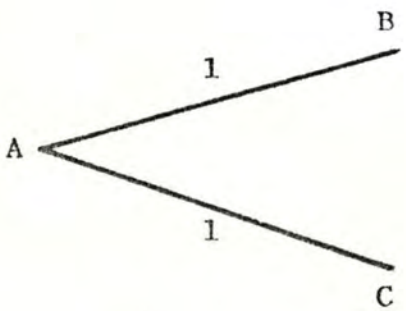
Lors d'un premier examen, le graphe partiel des relations de ce programme s'avère être :



Nous serions tenter de pondérer les arêtes comme suit :



tenant compte ainsi de la fréquence apparente des appels des blocs B et C par le bloc A.
Mais il est possible que par le jeu des données et du code environnant que la probabilité de passage dans la boucle soit de 0.01 tandis que la probabilité de passage sur l'instruction d'appel au bloc C soit égale à 1.
La pondération des arêtes telle que nous l'avions proposée ne s'explique plus, nous serions certainement plus proche de la réalité en proposant la pondération suivante :



Des exemples multiples pourraient ainsi être présentés démontrant les inconvénients de cette méthode.
Les améliorations peu importantes qu'elle apporte doivent toutefois être considérées pour son coût réduit.
Le mécanisme de pagination dépendant du comportement réel du programme, c'est à partir de mesures relevées dynamiquement que nous allons calculer l'affinité entre les blocs des programmes.

2.5.3. Calcul dynamique de l'affinité.

2.5.3.1. Calcul dynamique de l'affinité basé sur l'emplacement des références les unes par rapport aux autrss dans la chaîne.

2.5.3.2. Calcul dynamique de l'affinité basé sur le temps.

2.5.3.3. Echantillonnage de la chaîne de référence

2.5.3.1. Calcul dynamique de l'affinité basé sur l'emplacement des références les unes par rapport aux autres dans la chaîne.

Il est possible de calculer l'affinité entre les différents blocs d'un programme en ne tenant compte que des références contigues d'un bloc à un autre.
Ce calcul se fera au moyen de la fonction

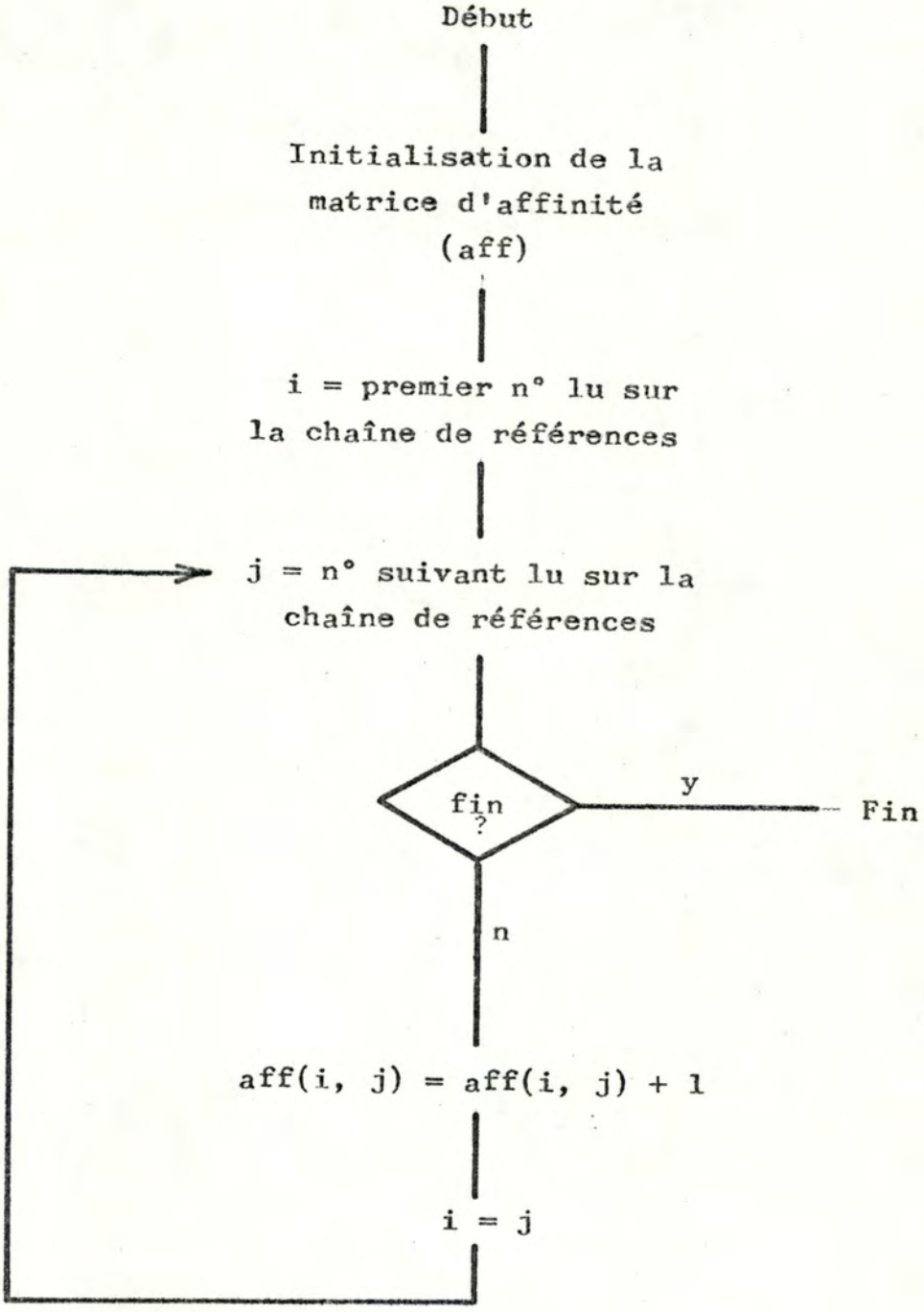
$f(n) = k(C - n)$

avec $k = 1$,

$C = 1$, et dans ce cas : $n = 0$

Pour se faire, on examinera la chaîne de références constituées d'identificateurs de blocs (noms ou n°) au travers d'une fenêtre ne permettant de voir que deux identificateurs contigus de la chaîne.
Nous comptabiliserons ces références dans une matrice d'affinité et déplacerons ensuite la chaîne d'un identificateur vers la gauche de façon à faire apparaître la référence suivante.

Dans le cas d'une chaîne de référence constituée de n° de blocs (il est toujours possible de se ramener à ce cas) l'algorithme de calcul de l'affinité est le suivant :



La chaîne $r_a\ r_b\ r_c\ r_b\ r_d\ r_b\ r_d\ r_b\ r_a$

sera par exemple traitée comme suit :

$r_a\ r_b$	$r_c\ r_b\ r_d\ r_b\ r_d\ r_b\ r_a$	$aff(r_a, r_b) = aff(r_a, r_b)+1$
$r_b\ r_c$	$r_b\ r_d\ r_b\ r_d\ r_b\ r_a$	$aff(r_b, r_c) = aff(r_b, r_c)+1$
$r_c\ r_b$	$r_d\ r_b\ r_d\ r_b\ r_a$	$aff(r_c, r_b) = aff(r_c, r_b)+1$
$r_b\ r_d$	$r_b\ r_d\ r_b\ r_a$	$aff(r_b, r_d) = aff(r_b, r_d)+1$
$r_d\ r_b$	$r_d\ r_b\ r_a$	$aff(r_d, r_b) = aff(r_d, r_b)+1$
$r_b\ r_d$	$r_b\ r_a$	$aff(r_b, r_d) = aff(r_b, r_d)+1$
$r_d\ r_b$	r_a	$aff(r_d, r_b) = aff(r_d, r_b)+1$
$r_b\ r_a$		$aff(r_b, r_a) = aff(r_b, r_a)+1$

FENETRE

Si la chaîne comporte N identificateurs, le nombre de comptabilisation à effectuer est (N-1) . On peut donc estimer que le temps de calcul des affinités est proportionnel à N.

2. Un calcul similaire permet de tenir compte des références espacées,
Nous examinerons cette fois la chaîne de références au travers d'une fenêtre laissant voir m(m>2) identificateurs de bloc contigus. La comptabilisation de l'affinité sera réalisée comme suit pour chaque fenêtre de taille n :

$$r_{a_1} \ldots\ldots \boxed{r_{a_i} \ldots\ldots r_{a_{i+1}} \ldots\ldots r_{a_{i+m-1}}} \ldots\ldots\ldots r_{a_n}$$

$$aff(r_{a_i}, r_{a_{i+1}}) = aff(r_{a_i}, r_{a_{i+1}}) + b_1$$

$$aff(r_{a_i}, r_{a_{i+2}}) = aff(r_{a_i}, r_{a_{i+2}}) + b_2$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$aff(r_{a_i}, r_{a_{i+m-1}}) = aff(r_{a_i}, r_{a_{i+m-1}}) + r_{b_{m-1}}$$

Les incréments r_{b_1}, r_{b_2}, ..., $r_{b_{m-1}}$ sont choisis en fonction de ce qui a été convenu au paragraphe 2.3.1.
Ils vérifieront en tout cas les inégalités suivantes :

$$r_{b_1} \geqslant r_{b_2} \geqslant \ldots\ldots\ldots\ldots \geqslant r_{b_{m-1}}$$

On choisit souvent la fonction :

$$f(n) = k(C - n) \qquad \text{avec} \quad k = 1, \quad C = m - 1$$

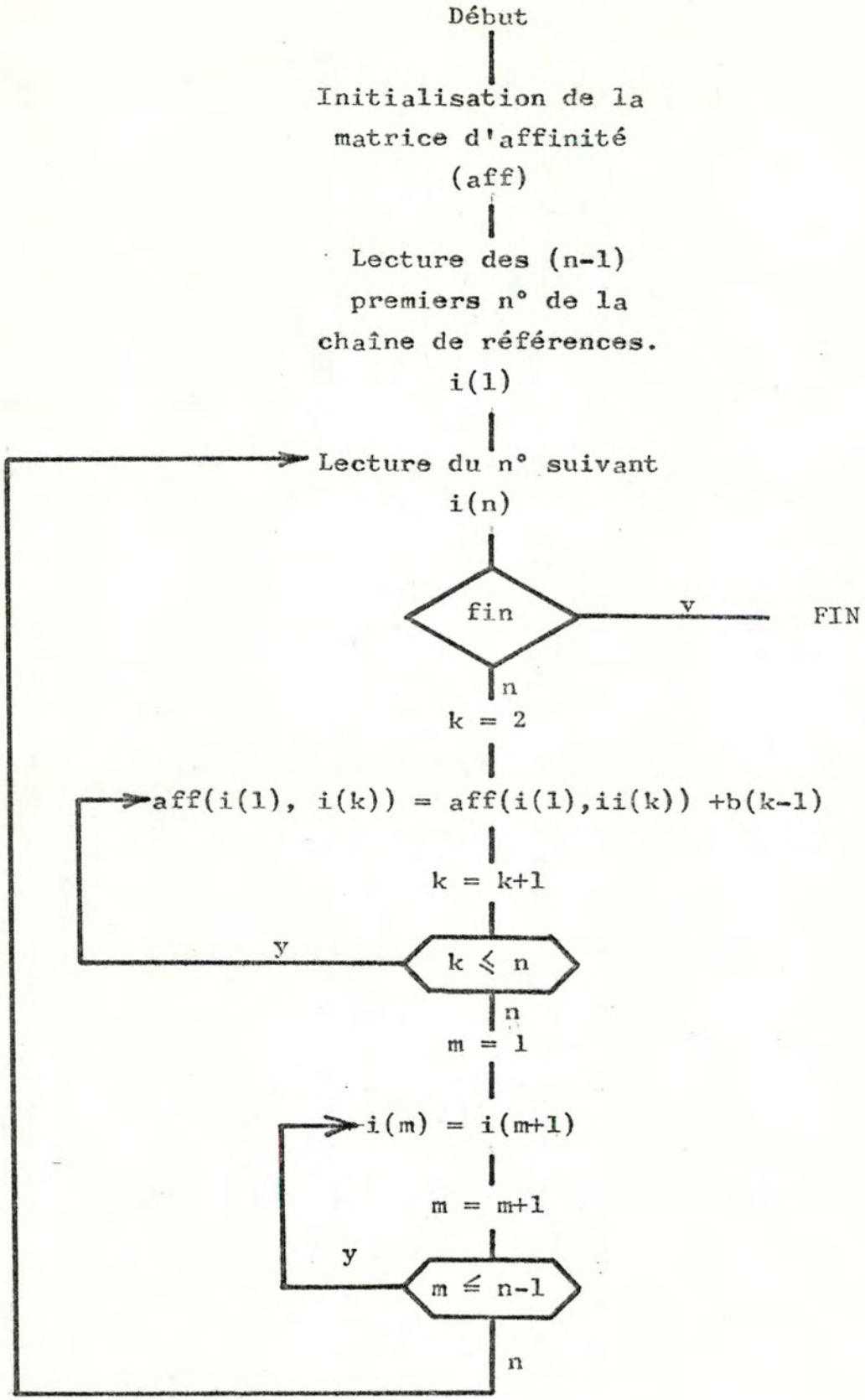
La chaîne de références est comme précédemment déplacée d'une référence vers la gauche après chaque groupe de comptabilisations correspondant à une fenêtre.

Si la chaîne comporte N identificateurs et si la fenêtre est de taille n ($n \ll N$) le temps de calcul de l'affinité est proportionnel à (N-n+1) et à (n-1) soient le nombre de déplacements de la fenêtre et le nombre de comptabilisations à effectuer pour chaque fenêtre.
Le temps de calcul sera donc de l'ordre de N *(n-1).
Pour n = 2 on retrouvera bien la valeur déduite au paragraphe précédent.

Soient une chaîne de références constituée de n° de bloc

n : la taille de la fenêtre
b : le vecteur des incréments

l'algorithme de calcul de l'affinité peut être représenté par l'organigramme suivant : (t.s.v.p.)

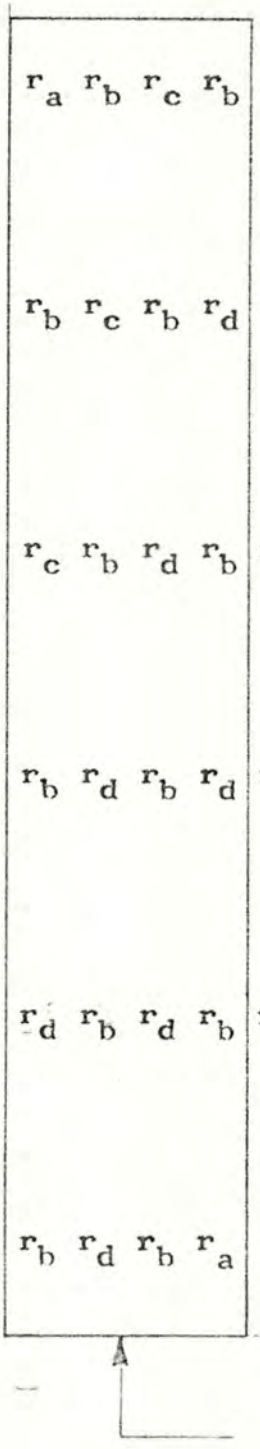

Début
Initialisation de la matrice d'affinité (aff)
Lecture des (n-1) premiers n° de la chaîne de références. i(1)
Lecture du n° suivant i(n)
fin
v
FIN
n
k = 2
aff(i(1), i(k)) = aff(i(1),ii(k)) +b(k-1)
k = k+1
y
k ⩽ n
n
m = 1
i(m) = i(m+1)
m = m+1
y
m ⩽ n-1
n

Exemple : la chaîne

$r_a\ r_b\ r_c\ r_b\ r_d\ r_b\ r_d\ r_b\ r_a$

sera traitée comme suit en tenant compte d'une fenêtre de taille 4 et des incréments

$r_{b_1} = 3$ $\quad r_{b_2} = 2$ $\quad r_{b_3} = 1$



$r_a\ r_b\ r_c\ r_b$	$r_d\ r_b\ r_d\ r_b\ r_a$	$aff(r_a, r_b) = aff(r_a, r_b)+3$
		$aff(r_a, r_c) = aff(r_a, r_c)+2$
		$aff(r_b, r_c) = aff(r_b, r_c)+1$
$r_b\ r_c\ r_b\ r_d$	$r_b\ r_d\ r_b\ r_a$	$aff(r_b, r_c) = aff(r_b, r_c)+3$
		$aff(r_b, r_b) = aff(r_b, r_b)+2$
		$aff(r_b, r_d) = aff(r_b, r_d)+1$
$r_c\ r_b\ r_d\ r_b$	$r_d\ r_b\ r_a$	$aff(r_c, r_b) = aff(r_c, r_b)+3$
		$aff(r_c, r_d) = aff(r_c, r_d)+2$
		$aff(r_c, r_b) = aff(r_c, r_b)+1$
$r_b\ r_d\ r_b\ r_d$	$r_b\ r_a$	$aff(r_b, r_d) = aff(r_b, r_d)+3$
		$aff(r_b, r_b) = aff(r_b, r_b)+2$
		$aff(r_b, r_d) = aff(r_b, r_d)+1$
$r_d\ r_b\ r_d\ r_b$	r_a	$aff(r_d, r_b) = aff(r_d, r_b)+3$
		$aff(r_d, r_d) = aff(r_d, r_d)+2$
		$aff(r_d, r_b) = aff(r_d, r_b)+1$
$r_b\ r_d\ r_b\ r_a$		$aff(r_b, r_d) = aff(r_b, r_d)+3$
		$aff(r_b, r_b) = aff(r_b, r_b)+2$
		$aff(r_b, r_a) = aff(r_b, r_a)+1$

2.5.3.2. Calcul dynamique de l'affinité basé sur le temps.

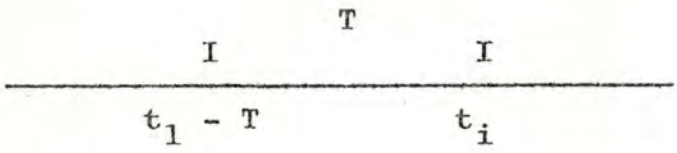
2.5.3.2.1. Extension des méthodes précédentes
2.5.3.2.2. Utilisation des working-sets
2.5.3.2.3. Méthodes des working-sets critiques

2.5.3.2.1. Extension des méthodes précédentes

Il serait possible d'établir un calcul de l'affinité basé sur le temps séparant les références d'une manière quasiment identique à ce qui a été fait au paragraphe précédent. Il nous suffirait de définir une fenêtre dont la taille serait cette fois exprimée en unité de temps. Nous ne développerons pas cette possibilité, nous décrirons plutôt deux algorithmes dus à FERRARI qui correspondent à une approche quelque peu différente du calcul de l'affinité.

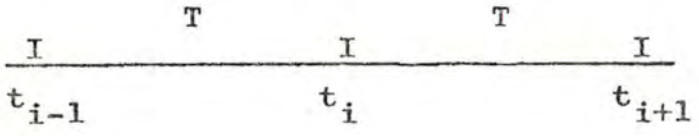
2.5.3.2.2. Utilisation des working-sets /17/

Il est nécessaire que nous introduisions ici une nouvelle notion : celle de working-set. /13/
Le working-set au temps t_i d'un programme correspond à l'ensemble des blocs référencés par ce programme durant l'intervalle de temps $(t_i - T, t_i)$ où T est le paramètre du working-set ou encore la taille de la fenêtre. (Si le temps séparant chaque référence est constant, la fenêtre utilisée au paragraphe précédent permet l'introduction de cette notion.)



Pour ce qui suit nous considérerons que T est constant et égal à $t_{i-1} - t_i$ quel que soit i.

1. Considérons deux intervalles consécutifs (t_{i-1}, t_i) et (t_i, t_{i+1}) de durée T :



Nous estimerons le working-set correspondant à la seconde période par le working-set de la période précédente soit $W(t_i, T)$. Toutefois, le working-set de la seconde période est $W(t_{i+1}, T)$ et non $W(t_i, T)$.

L'ensemble $M(t_i, t_{i+1}) = W(t_{i+1}, T) \setminus W(t_i, T)$ constitue l'ensemble des blocs manquants : c'est à dire les blocs qui ont été référencés pendant la période (t_i, t_{i+1}) mais qui n'étaient pas inclus dans l'information estimée.
Chacun de ces blocs est susceptible de provoquer un défaut de page (s'ils se trouvent tous dans des pages distinctes). Chaque page manquante sera ajoutée à celles se trouvant déjà en mémoire et accroîtra la taille du working-set. Le programme sera suspendu s'il n'y a pas suffisament d'espace mémoire disponible.

L'ensemble $E(t_i, t_{i+1}) = W(t_i, T) \setminus W(t_{i+1}, T)$ représente l'ensemble des blocs en excès : ce sont les blocs qui font partie de l'information active estimée mais qui ne sont pas référencés pendant l'intervalle de temps (t_i, t_{i+1}), ils occupent donc la mémoire en pure perte.

La prédiction de l'information active serait parfaite si les ensembles M et E étaient vides, donc si $W(t_{i+1}, T)$ était égal à $W(t_i, T)$.

En conséquence le groupement de deux blocs dans une page peut être utile dans les deux cas qui suivent :

- l'un des deux blocs fait partie du working-set estimé, l'autre étant un bloc manquant, soient les blocs p et q tels que :

$$\begin{cases} p \in W(t_i, T) \\ q \in M(t_i, t_{i+1}) \end{cases}$$

  car la page contenant p et q ne sera plus manquante et la référence à q ne causera, de ce fait plus de défaut de page.

- l'un des deux bloc est en excès, l'autre fait partie du working-set à la même période, soient les blocs r et s tels que :

$$\begin{cases} r \in E(t_i, t_{i+1}) \\ s \in W(t_{i+1}, T) \end{cases}$$

  car la page contenant r ne sera plus en excès, l'on ne gardera donc pas de page inutile en mémoire pendant la période (t_i, t_{i+1})

l'algorithme se présentera donc comme suit :
pour chaque paire d'intervalles consécutifs on calculera les ensembles M et E et l'on réalisera les comptabilisations qui suivent :

a) pour tous blocs p et q tels que

$p \in W(t_i, T)$ et $q \in M(t_i, t_{i+1})$

$aff(p, q) = aff(p, q) + 1$

b) pour tous blocs r et s tels que

$r \in E(t_i, t_{i+1})$ et $s \in W(t_{i+1}, T)$

$aff(r, s) = aff(r, s) + 1$

2.5.3.2.3. Méthode des working-sets critiques /18/

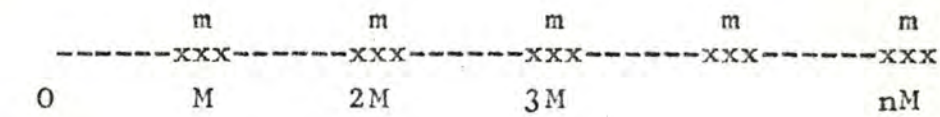
Un working-set critique est défini par FERRARI comme étant un working-set W(t, T) tel que la prochaine page référencée après l'instant t n'appartient pas à W(t, T). Cette prochaine référence est appelée page critique.
La matrice d'affinité est construite de telle façon que aff(i, j) représente le nombre de working-sets critiques de la chaîne de références ayant le bloc i comme référence critique et contenant le bloc j.

2.5.3.3. Calcul dynamique de l'affinité par échantillonnage de la chaîne de références.

Les calculs dynamiques d'affinités décrits jusqu'ici présentent le désavantage d'être particulièrement coûteux du fait que leur temps de calcul est proportionnel à la taille de la chaîne de références, taille qui est toujours très grande. Les essais que nous avons réalisés nous permettent d'affirmer que cette chaîne voit sa taille s'accroitre en moyenne de 1.500 identificateurs par seconde CPU écoulée et ce en ne tenant compte que des transferts de contrôle. Une restructuration ne s'appliquant qu'à des programmes importants, on comprendra que cette chaîne puisse compter plusieurs dizaines voire centaines de milliers d'identificateurs.

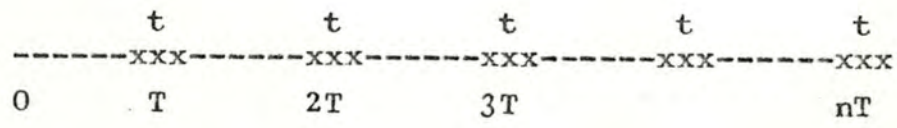
En vue de réduire ces coûts, il est possible de ne pas utiliser l'entièreté de la chaîne de références en vue de l'établissement de la matrice d'affinité, par exemple en échantillonnant cette chaîne, soit à partir des références seules, soit à partir des temps CPU qu'il est possible de collecter à chaque transition.

Dans le premier cas, nous considérerons des échantillons de taille fixe comportant m références, les échantillons étant constitués toutes les M références.



(réf.)

Dans le second cas, les échantillons seront constitués des références survenues pendant des intervalles de temps t à chaque période T.



Les méthodes de calcul de l'affinité examinée précédemment peuvent encore s'appliquer aux échantillons. Il en existe d'autres, plus spécialement adaptées. Nous allons en examiner deux :

1) le calcul suivant est réalisé pour chaque échantillon :

$$aff(i,j) = aff(i,j) + \delta_i * \delta_j$$

où δ_i | = 1 si le bloc i se trouve dans l'échantillon.
= 0 sinon.

L'affinité entre deux blocs est donc égale au nombre d'échantillons comportant ces deux blocs.

2) pour chaque échantillon on calcule le nombre d'occurences de chaque identificateur de bloc : soit n_i le nombre d'occurences de l'identificateur i.

La matrice est ensuite mise à jour comme suit :

$$aff(i,j) = aff(i,j) + n_i * n_j$$

L'échantillonnage de la chaîne de références permet de réduire le coût du calcul de l'affinité dans une proportion m/M ou t/T suivant que l'on échantillonne cette chaîne par rapport aux références ou par rapport au temps.

CHAPITRE III : RESTRUCTURATION

3.1. Introduction
3.2. Algorithmes tenant compte de la pagination de la mémoire.
3.3. Algorithmes indépendants de la pagination.

3.1. Introduction.

Après avoir scindé le programme en blocs, calculé les affinités entre ceux-ci, nous nous proposons de décrire ici quelques algorithmes qui nous permettrons de restructurer le programme.
La notion de bloc apparaissant dans ce chapitre est plus restrictive que celle que nous avons décrite précédemment : un bloc représentera dorénavant un ensemble de données ou d'instructions relogeable.
Il serait en outre souhaitable que la taille des blocs soit en moyenne très inférieure à la taille d'une page.

Les différents algorithmes de restructuration que nous allons examiner se répartissent en deux classes:

1. les algorithmes qui tiennent compte de la pagination de la mémoire en vue de minimiser le nombre de blocs s'étendant de part et d'autre d'une (de) frontière(s) de page : tout bloc dont la taille est inférieure à une page, sera toujours mémorisé dans une et une seule page. Ces algorithmes ne donnent toutefois pas toujours des résultats satisfaisants par suite des fractions de pages inutilisées qu'ils laissent à la fin de la plupart de celles-ci. Aussi avons nous envisagé la possibilité de densifier les programmes restructurés obtenus par ces techniques.

2. Nous donnerons par la suite quelques indications concernant la restructuration des programmes de façon indépendante de la pagination.

3.2. Algorithmes tenant compte de la pagination de la mémoire.

3.2.1. Algorithme de RYDER.
3.2.2. Algorithme de HATFIELD et GERALD
3.2.3. Algorithme de MASUDA.
3.2.4. Densification.

3.2.1. Algorithme de RYDER. /27/

Cet algorithme se décompose en deux phases distinctes
1. une simplification de la matrice d'affinité,
2. une exploitation de la matrice ainsi simplifiée en vue d'obtenir des listes de chargement.
   Une liste de chargement est constituée d'une suite d'identificateurs de bloc, correspondant aux blocs devant être chargés dans une même page, ou dans un même groupe de pages.

Phase I : Simplification de la matrice d'affinité.

1. Tout bloc dont la taille est supérieure à une page sera toujours mémorisé à partir d'une frontière de page.

Le programmeur connaîtra ainsi l'emplacement des frontières de page à l'intérieur des blocs qu'il rédige. Il pourra par exemple éviter de placer une boucle de part et d'autre d'une frontière.

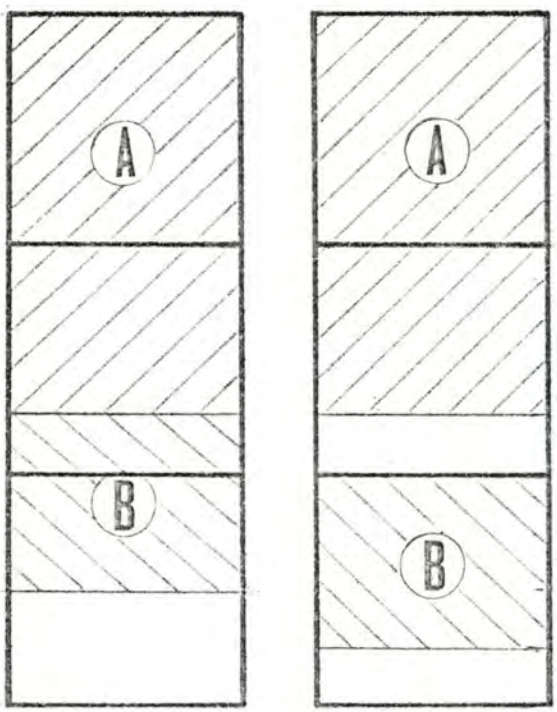
Règle : Les affinités entre blocs dont la taille est supérieure à une page seront annulées.

2. Tout bloc dont la taille est un multiple exact de la taille d'une page ne pourra être groupé avec aucun autre bloc.

Règle : Les affinités correspondant aux blocs dont la taille est un multiple exact de la taille d'une page seront annulées.

3. Tout bloc dont la taille est supérieure à une page pourra être groupé avec un bloc dont la taille est inférieure à une page si le groupement requiert moins de pages que les blocs n'en requéraient de façon séparée.

Par exemple un bloc de 1 3/4 pages ne sera jamais groupé avec un bloc de 3/4 page. Le groupe occupe en effet 3 pages, soit le même nombre de pages que les blocs séparés.



Ceci implique qu'un bloc ne s'étendra jamais de part et d'autre d'une frontière de page sauf si sa taille est supérieure à une page.

Règle : Les affinités entre blocs dont la somme des "tailles modulo la taille d'une page" est supérieure à une page seront annulées.

4. Les blocs ayant une affinité très faible seront placés, non plus en fonction de cette affinité, mais en fonction d'une utilisation aussi dense que possible de la mémoire.

Règle : Toutes les affinités inférieures à une certaine constante seront annulées.

Phase II : Exploitation de la matrice simplifiée.

Etape 1. : Rechercher et annuler l'élément de valeur maximale de la matrice d'affinité. Si la matrice est nulle, la restructuration est terminée, sinon trois situations peuvent se présenter :

1. aucun des deux blocs correspondants à l'élément maximum découvert ne font partie d'une liste de chargement.
(c'est certainement le cas lors du premier passage dans l'étape 1.).
Règle : aller à l'étape 2.

2. Les deux blocs font déjà partie de deux listes de chargement différentes.
Règle : aller à l'étape 3.

3. un et un seul des deux blocs fait déjà partie d'une liste de chargement.
Règle : aller à l'étape 4.

Etape 2. : Les simplifications précédentes de la matrice d'affinité garantissent que les deux blocs concernés peuvent être groupés dans une ou plusieurs pages. Une liste de chargement est constituée pour ces deux blocs. L'espace restant dans la page (ou dans la dernière page) est calculé en considérant la somme des tailles des blocs modulo la taille d'une page, somme que l'on soustrait de la taille de la page.
Si cette quantité est inférieure à la taille de tous les blocs non encore groupés, la liste est complète, la matrice d'affinité peut-être simplifiée en tenant compte du

fait que les blocs appartenant à une liste complète ne doivent plus être considérés.

Aller à l'étape 1.

Etape 3. : Les deux blocs concernés font déjà partie de deux listes de chargement distinctes.
Les affinités existant entre les blocs des deux listes sont annulées.
Si les deux listes comportent chacunes un bloc dont la taille est supérieure à la taille d'une page ou si la somme des tailles des blocs des deux listes modulo la taille d'une page est supérieure à la taille d'une page les deux listes ne peuvent être regroupées en une seule.

Aller à l'étape 1.

Sinon les deux listes sont groupées. L'espace restant dans la nouvelle liste ainsi formée est calculé, s'il est inférieur à la taille de tout bloc non encore groupé, la liste est complète, tous les éléments de la matrice correspondants aux blocs de la liste sont annulés.

Aller à l'étape 1.

Etape 4. : Toutes les affinités entre le nouveau bloc et les blocs de la liste concernée sont annulées.
Si la taille du nouveau bloc est supérieure à la taille d'une page alors qu'il existe déjà dans la liste un bloc dont la taille est également supérieure à la taille d'une page
ou si la taille modulo la taille d'une page

du nouveau bloc est supérieure à l'espace libre de la liste, le nouveau bloc ne pourra être inclu dans cette liste.

Aller à l'étape 1.

Sinon le nouveau bloc est inclus dans la liste.
L'addition d'un nouveau bloc à la liste nous oblige de recalculer l'espace libre de cette liste. Si cet espace est inférieur à la taille de tout bloc non groupé, la liste est complète, tous les éléments correspondants aux blocs de la liste sont annulés.

Aller à l'étape 1.

Quand la recherche des listes de chargement effectuée par l'algorithme de RYDER est terminée, il reste généralement un certain nombre de blocs ne faisant partie d'aucune de ces listes.
Ces blocs seront placés de façon à rendre l'occupation de la mémoire aussi dense que possible, on les placera donc dans les espaces inoccupés situés en fin des pages correspondants à des listes non complètes et éventuellement si cela est nécessaire dans des pages additionnelles.
Diverses méthodes peuvent être mises en oeuvre pour se faire, il se peut entre autre qu'une solution apparaisse directement sans nécessiter aucun calcul. Ce ne sera toutefois pas le cas en général, aussi allons nous développer une méthode heuristique permettant d'obtenir une solution. /08/

Notre problème peut s'énoncer comme suit :
comment placer b blocs de tailles variables dans $t = t_1 + t_2$ espaces comprenant t_1 espaces de tailles variables (correspondants aux pages partiellement remplies) et t_2 espaces de taille fixe (pages supplémentaires) de façon à occuper aussi peu de pages supplémentaires que possible ?

Nous pouvons influencer le résultat par deux et seulement deux paramètres :

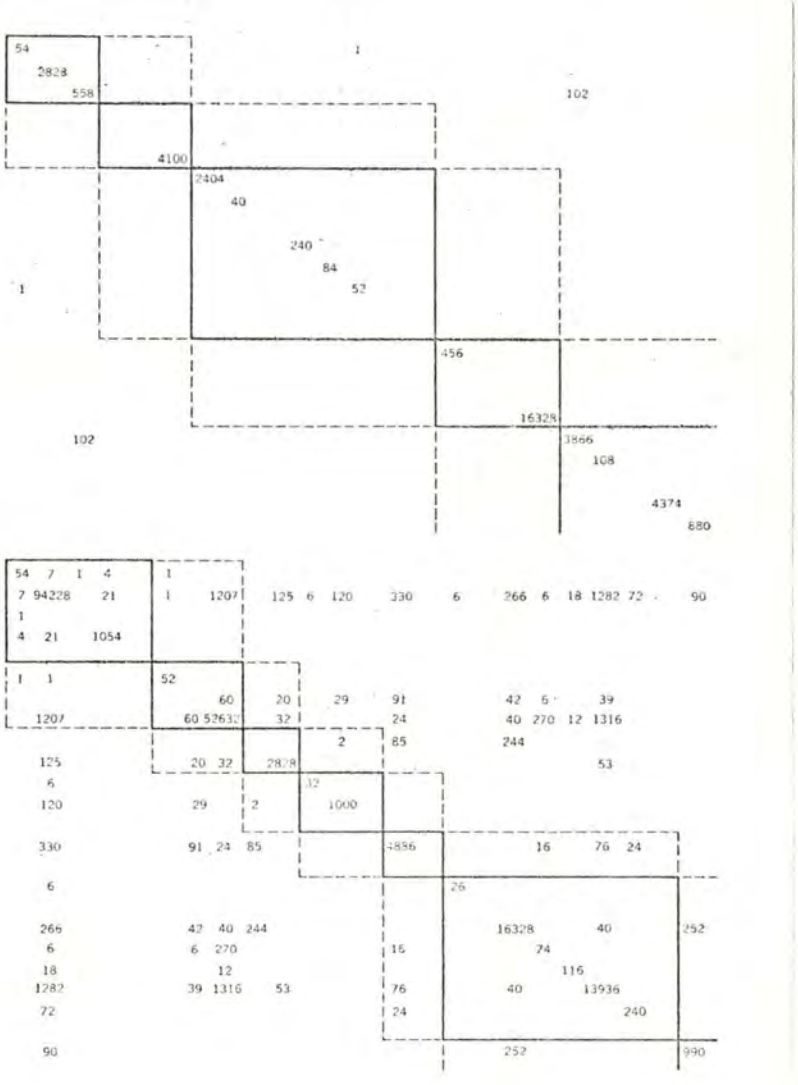
- l'ordre du choix des blocs,
- l'ordre du remplissage des trous.

Pendant le remplissage, si les blocs de tailles élevées sont laissés pour la fin, il sera sans doute difficile de les placer, on commencera donc par placer les blocs de tailles maximales.
Du fait que nous souhaitons occuper un nombre minimum de pages, nous essaierons en premier lieu de remplir les pages partiellement remplies, et parmi celles-ci celles qui présentent les plus petits espaces libres.

3.2.2. Algorithme de HATFIELD et GERALD. /20/

Correspondant à chaque arrangement des blocs d'un programme, existe un réarrangement des lignes et des colonnes de la matrice d'affinité. Il est évident que les arrangements qui se réflètent dans la matrice d'affinité par la présence des plus grands éléments de celle-ci près de la diagonale principale sont les meilleurs.

Si l'on admet que chaque page ne peut contenir qu'un nombre entier de blocs, on associera à toute page contenant s blocs une sous-matrice s x s. Un bon arrangement contiendra les éléments de valeurs maximales de la matrice d'affinité dans ces sous-matrices, comme le montrent les deux exemples suivants :



A partir d'une matrice d'affinité donnée, nous rechercherons donc une méthode de réarrangement de ses lignes et colonnes qui place les plus grands éléments de cette matrice dans des sous matrices carrées situées sur la diagonale principale.
Ces sous-matrices n'auront pas toutes la même taille étant donné que les blocs sont de taille variable.
On pourra s'attendre à trouver des nombres de blocs différents dans des pages différentes.

La méthode proposée peut-être visualisée comme suit :

- les différents blocs du programme sont représentés par les noeuds de poids nul d'un treillis.
- ces noeuds sont reliés entre eux par des ressorts dont la force de rappel est proportionnelle à l'affinité existant entre les blocs qu'ils représentent
- de plus chaque noeud est relié au sol par un ressort dont la force de rappel est g.

Exemple : Soit un programme constitué de quatre blocs A, B, C et D et dont la matrice d'affinité est la suivante :

	A	B	C	D
A				
B	3			
C	1	2		
D	4	0	5	

La représentation statique en est donnée à la figure ci-contre.

Pour connaître les blocs devant se trouver dans la même page que le noeud i, nous soulèverons ce noeud jusqu'à ce que ce dernier élève avec lui suffisament d'autres noeuds tels que les blocs qu'ils représentent puissent remplir une page.
Pour chaque noeud i élevé à une hauteur h_i nous souhaiterons donc connaître les hauteurs atteintes par les noeuds qu'il entraîne.
Ceci constitue un problème de statique qui peut être résolu en minimisant l'énergie du système ainsi construit. Les hauteurs relatives des noeuds par rapport au noeud i sont données par la ième ligne de l'inverse d'une matrice D construite comme suit :

$$d(i, j) = aff(i, j) \text{ pour } i \neq j$$

$$d(i, i) = \sum_{j=1}^{n} aff(i, j) + g$$

g étant choisi de l'ordre de 2n

n étant le nombre de blocs /20/

Les lignes de D^{-1} sont alors comparées en examinant pour chaque ligne i, l'ensemble des noeuds proches du noeud i et l'ensemble des autres noeuds. L'ensemble des noeuds proches de i étant défini comme l'ensemble des noeuds représentants des blocs pouvant se trouver dans la même page que le bloc i.

La meilleure ligne est choisie en comparant les hauteurs relatives au noeud i des noeuds proches de celui-ci.

soit si α_i représente l'ensemble des noeuds proche de i on comparera les sommes

$$\sum_{j \in \alpha_j} dij \qquad \forall\, i$$

La ligne correspondant à la somme minimale sera choisie.

Les candidats de la meilleur ligne sont retenus et la matrice D^{-1} est réduite des lignes et colonnes correspondant à ceux-ci.
Le processus de sélection est alors itéré jusqu'à ce que tout les blocs soient placés.

3.2.3. Algorithme de MASUDA. /22/

Les techniques d'essaimage peuvent être appliquées à notre problème de réorganisation de programmes.

La structure générale de l'algorithme que nous utiliserons est la suivante :

Etape 1. : Nous débutons avec n essaims, chacun étant constitué d'un et d'un seul bloc du programme.
Nous numéroterons ces essaims de 1 à n.

Etape 2. : Nous rechercherons les deux essaims ayant la plus forte affinité, soient les essaims p et q (p>q) et aff(p, q) leur affinité.
Si cette affinité est inférieure à un nombre donné, le regroupement est terminé.

Etape 3. : nous réduirons le nombre d'essaims de une unité en groupant les deux essaims p et q en un nouvel essaim q.
Nous mettrons à jour la matrice d'affinité de manière à refléter les nouvelles affinités entre l'essaim q et les autres.
La ligne et la colonne relative à l'essaim p seront annulées.
Nous mémoriserons l'identité des essaims p et q ainsi que aff(p, q).
Aller à l'étape 2.

Cette structure permet d'implémenter différents algorithmes suivant le choix du mode de mise à jour de la matrice d'affinité à l'étape 3.

MASUDA et autres /22/ ont développé une méthode de mise à jour tenant compte des tailles des essaims.

L'affinité entre deux essaims p et q est définie comme suit :

$$aff(p, q) = \frac{1}{s_p + s_q} \sum_{i \in p} \sum_{j \in q} aff(i, j)$$

ou s_p et s_q représentent la taille des essaims p et q (la somme des tailles des blocs qui les constituent).

l'affinité entre blocs doit bien évidemment être corrigée comme suit :

$$aff(i, j) = \frac{aff(i, j)}{(s_i + s_j)}$$

s_i représentant la taille du bloc i.

3.2.4. Densification.

Tous les blocs étant placés, il reste le problème important des pages partiellement remplies. Si les blocs ne peuvent s'étendre de part et d'autre de frontières de page, il restera toujours des espaces inutilisés dans les pages; dans le cas contraire il est possible de densifier le programme restructuré de façon à ne plus laisser aucun espace inutilisé. Les expériences réalisées montrent les résultats encourageant de cette densification.

Si les blocs peuvent traverser les frontières de page, un choix correct des groupes adjacents devra être réalisé. Un bloc situé sur une frontière de page requerera probablement lors de son exécution les deux pages sur lesquelles il s'étend.
En vue de s'assurer que les deux pages soient en mémoire chaque fois que le bloc frontière est utilisé nous essaierons de placer les uns à la suite des autres les groupes ayant les affinités les plus fortes.
Nous définirons l'affinité entre deux groupes A' et B' comme la somme des affinités des blocs d'un des groupes vis-à-vis des blocs de l'autre groupe, soit :

$$aff(A', B') = \sum_{\substack{i \in A' \\ j \in B'}} aff(i, j)$$

Notre problème consiste à découvrir un ordre des groupes de telle manière que la somme des affinités entre les groupes contigus soit maximale.
Il se ramène facilement au problème du voyageur de commerce si l'on considère un circuit englobant les différents groupes.

3.3. Algorithmes indépendants de la pagination.

Ces algorithmes diffèrent en fait très peu de l'algorithme de MASUDA. Ils sont basés sur la même structure que celle décrite au paragraphe 3.2.3. Seule est modifiée la méthode de mise à jour de la matrice d'affinité qui cette fois ne dépend plus de la taille des essaims.

L'affinité entre deux essaims p et q peut être par exemple définie comme suit :

a) méthode de l'affinité maximale

$$aff(p, q) = \max_{i \in p, j \in q} aff(i, j)$$

b) méthode de l'affinité minimale

$$aff(p, q) = \min_{i \in p, j \in q} aff(i, j)$$

pour $aff(i, j) \neq 0$

CHAPITRE IV : OPTIMISATION DES ALGORITHMES

4.1. Introduction.
4.2. Recherche de l'élément de valeur maximale dans la matrice d'affinité.
4.3. Optimisation de l'accès à la table des identificateurs de blocs.

4.1. Introduction

Les algorithmes que nous avons décrits dans le chapitre qui précède sont inutilisables pratiquement si certaines de leurs fonctions ne sont pas optimisées. En effet, le coût d'une restructuration peut-être tel qu'il ne soit pas économique de la réaliser. Aussi, croyons-nous qu'il est indispensable d'examiner quelques améliorations permettant de réduire considérablement le coût de la restructuration d'un programme. Ces améliorations portent sur la recherche de l'élément de valeur maximale de la matrice d'affinité et sur l'optimisation de l'accès à la table des symboles.

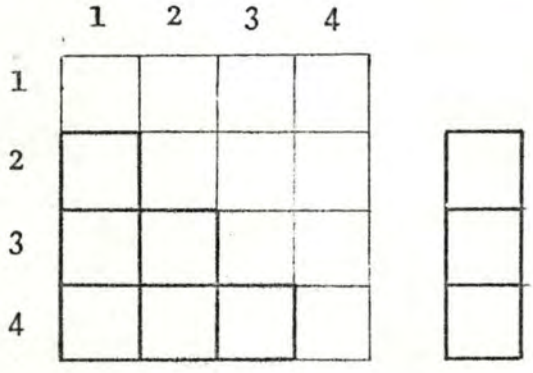
4.2. Recherche de l'élément de valeur maximale de la matrice d'affinité.

Dans tous les algorithmes de restructuration, il est nécessaire à chaque itération de rechercher l'élément de valeur maximale dans la matrice d'affinité.
Dans le cas d'un programme constitué de n blocs, la matrice d'affinité comporte $\frac{n * (n - 1)}{2}$ éléments.
Si la recherche du maximum est séquentielle, elle se réalisera au moyen de $\frac{n * (n - 1)}{2} - 1$ comparaisons.

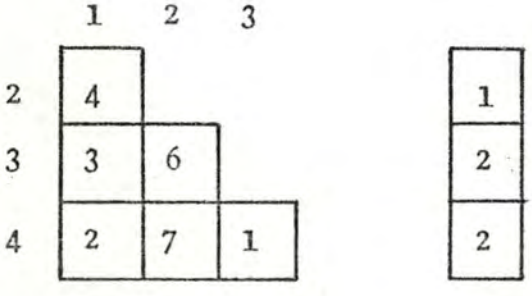
Pour un programme décomposé en une centaine de blocs ceci conduit à chaque itération à (9900/2 - 1) = 4949 comparaisons.

Nous allons essayer de réduire ce nombre et donc d'accélérer la recherche du maximum en nous basant sur le fait que la matrice est souvent peu modifiée d'une itération à l'autre.

Pour ce faire, nous utiliserons un vecteur de dimension n - 1, chacun des éléments de ce vecteur étant associé à une ligne de la matrice comme indiqué sur la figure. qui suit :



Après avoir constitué la matrice, nous rechercherons le maximum dans chacune des lignes et indiquerons le numéro de la colonne dans laquelle ce maximum apparait dans la case correspondante du vecteur :



Ceci requiert 0 + 1 + 2 + ... +(n - 2) comparaisons soient $\frac{(n - 2) * (n - 1)}{2}$ comparaisons.

Ceci fait, la recherche du maximum de la matrice ne doit plus se faire que sur les maxima de ligne que l'on connait et qui sont au nombre de (n - 1).

Etant donné qu'après avoir découvert le maximum de la matrice, on l'annule de façon à poursuivre le processus itérativement, il faut mettre à jour la case du vecteur correspondant à la ligne ou se trouvait ce maximum. Cette recherche du maximum de ligne comporte en moyenne (n - 2)/2 comparaisons.
Ce procédé permet donc de calculer à chaque itération (sauf la première et celles qui modifient une colonne de la matrice) le maximum de cette matrice en

(n - 2) + (n - 2)/2 = 3/2 * (n - 2) comparaisons.

Le tableau qui suit permet d'estimer la réduction obtenue dans différents cas :

Programme constitué de n blocs	Nombre de comparaisons	
	sans optimisation	avec optimisation
25	399	35
50	1224	72
100	4949	147
200	19900	297
400	79800	597

On peut montrer que la réduction du nombre de comparaisons est de l'ordre de n/3.
En effet :

$$\frac{\frac{n*(n-1)}{2} - 1}{\frac{3}{2}(n-2)} = \frac{(n*(n-1)) - 2}{3(n-2)} = \frac{O(n^2)}{O(3n)} = O(\frac{n}{3})$$

4.3. Optimisation de l'accès à la table des identificateurs de blocs.

Dans le cours du calcul de l'affinité entre les différents blocs du programme et en cours de restructuration, il est fréquemment fait accès à la table des identificateurs de bloc.
Cette table a comme argument les identificateurs de blocs et comme valeurs le n°, la taille, l'adresse et d'autres renseignements se rapportant à chacun des blocs.
Cette table étant très sollicitée il importe que les recherches que l'on y fait soient aussi rapides que possible.

La manière la plus simple de l'organiser consiste à ajouter les arguments dans l'ordre dans lequel ils arrivent. Une recherche consistera alors en une comparaison avec chaque entrée jusqu'à ce que l'on découvre le bon argument.
Pour une table constituée de n arguments ceci demande en moyenne n/2 comparaisons. Si n est grand, cette méthode est très inefficace.

La recherche sera plus efficiente si la table est ordonnée, nous pourrons alors utiliser une méthode de recherche tenant compte de ce fait, par exemple une recherche dichotomique.
Le nombre maximum de comparaisons est dans ce cas de $1 + \log_2 n$. /30/

Le tableau suivant montre clairement l'avantage d'une recherche dichotomique par rapport à une recherche séquentielle.

Taille de la table	Recherche séquentielle		Recherche dichotomique
	Nombre moyen de comparaisons	Nombre maximum de comparaisons	Nombre maximum de comparaisons
16	8	15	4
32	16	31	5
64	32	63	6
128	64	127	7
256	128	255	8
512	256	511	9

CHAPITRE V : RESTRUCTURATION AUTOMATIQUE

5.1. Introduction.
5.2. Importance de l'éditeur de liens.
5.3. Le triangle d'auto-adaptation
5.4. Fonctionnement de l'éditeur de liens, Possibilité d'y inclure des modules de restructuration.

5.1. Introduction.

Au stade auquel nous sommes arrivés, nous pouvons ressentir la nécessité d'un outil de restructuration purement automatique qui vienne s'incorporer à la chaîne habituelle de traitement des programmes : compilation - liaison - exécution.

Cet outil devrait occuper un emplacement privilégié dans cette chaîne, en effet, il devrait :

- avoir une connaissance globale du programme,
- être indépendant des langages de programmation,
- pouvoir agir sur l'implantation.

5.2. Importance de l'éditeur de liens.

Cet emplacement privilégié n'est autre que celui occupé par l'éditeur de liens statiques comme on peut le vérifier sur la chaîne de traitement qui suit :

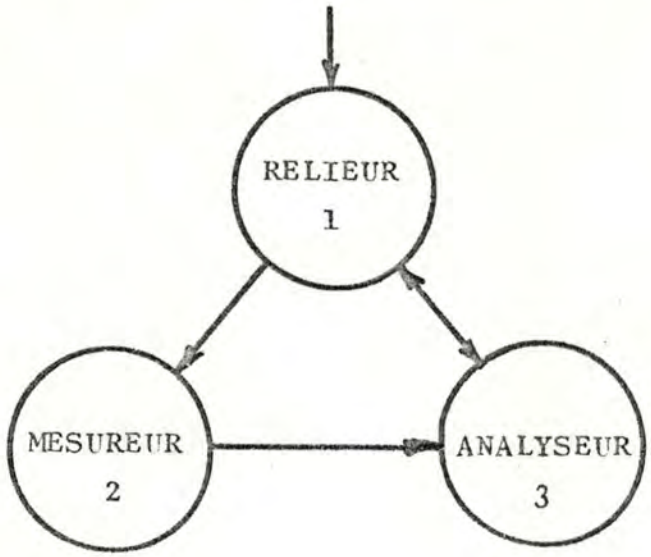
L'éditeur de liens dispose bien d'une connaissance globale du programme : tous les modules objets de ce dernier lui sont passés avec leurs tables. (Voir annexe II : structure du module objet IBM). Il est indépendant des langages de programmation étant donné qu'il n'utilise que des modules objets et peut agir sur l'implantation de ces modules, c'est lui qui transforme l'ensemble des modules objets en un module chargeable.
De plus, au niveau de l'éditeur de liens le programme est constitué de blocs bien adaptés à la restructuration.
Il est toutefois utile de prendre la précaution que chaque module objet construit par un compilateur corresponde à la notion de bloc décrite au chapitre 3. Il suffit pour ce faire de compiler chaque routine séparément de façon à ce que tous les modules objets ne comportent jamais qu'une seule section. Une section étant la forme traduite en langage objet d'une routine source.
L'éditeur de liens pourra réaliser les mesures nécessaires à une restructuration de deux manières différentes :

- à l'édition de liens, il met en évidence les liens de bloc à bloc et les mémorise dans une matrice d'affinité "statique".
- à l'exécution, les références de bloc à bloc sont comptabilisée par un module incorporé au programme par le relieur.

Ces caractéristiques de l'éditeur de liens nous permet d'introduire la notion de triangle d'auto-adaptation due à MORISSET.

5.3. Le triangle d'auto-adaptation.

Ce triangle d'auto-adaptation représente l'ensemble des moyens mis en oeuvre pour réaliser une restructuration.



Le point d'entrée du triangle est le relieur. C'est dans un fichier créé par le relieur que sont rangées toutes les informations relatives à la restructuration et en particulier les affinités statiques à la première édition de liens, dynamiques après exécution.

Il existe plusieurs cheminements possibles à l'intérieur de ce triangle :

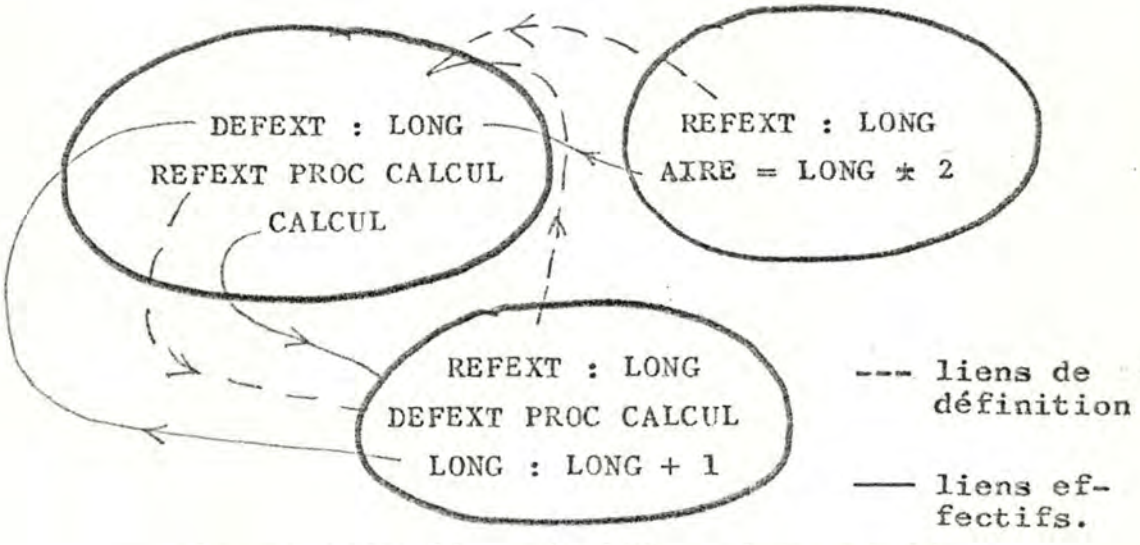
- à la première édition de liens en fin de phase 1., le relieur après avoir chargé la matrice d'affinité statique peut demander l'établissement d'une structure au module d'analyse, il tient compte de cette structure pour constituer le module chargeable auquel il aura incorporé un module de mesure.

- le programme est ensuite surveillé au cours de son exécution par le module de mesure (2) qui construit la matrice d'affinité dynamique.
- après cette première mesure, il fait appel au module d'analyse qui en déduit une nouvelle implantation (3).
- le relieur restructure le programme en créant un nouveau module chargeable auquel on pourra éventuellement encore incorporer un module de mesure de façon à pouvoir créer une implantation qui tiennent compte de plusieurs exécutions (données du programme.

5.4 Fonctionnement de l'éditeur de liens - Possibilité d'y inclure des modules de restructuration.

Si l'on considère un ensemble de modules-objets faisant partie d'un même programme, devant donc être exécutés ensemble, des liens doivent avoir été, ou être établis entre eux au moment de l'exécution. Comme précédemment nous supposerons que ces liens sont établis statiquement avant exécution.
Nous appelerons "externe" tout objet permettant ces liens entre les différents modules objets. Un externe est défini dans le module par son nom et est rendu accessible aux autres par l'intermédiaire de ce nom. Il acquiert ainsi une portée générale dans le programme.

Le module qui défini un tel objet le déclare en définition d'externe : DEFEXT. On trouvera par exemple dans le code source : DEFEXT : long.
Un module qui veut référencer un objet supposé défini en DEFEXT par un autre commence par le déclarer en référence d'externe : REFEXT. Le code source aura alors l'aspect suivant : REFEXT : Long;...;...;
Aire = long * 2;



Exemples de liens établis entre modules.

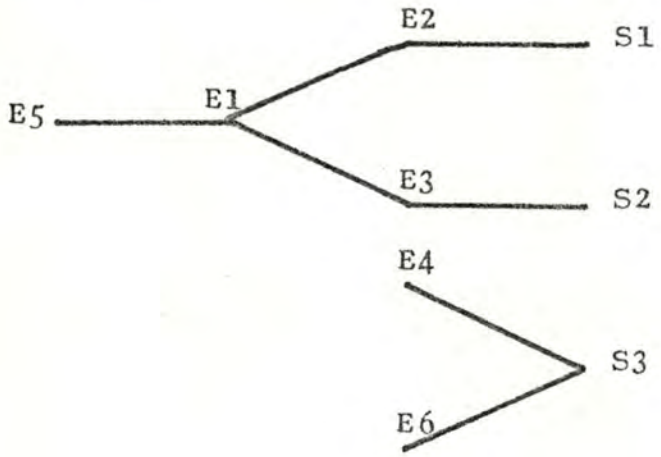
Tout externe est défini par rapport à un module objet. (par hypothèse chaque module objet ne comporte qu'une seule section).

Soit $G(X, \Gamma)$ le graphe de définition du programme :

- X : ensemble des objets du relieur
- Γ : relation interne à X telle que l'on aie $x \Gamma y$ si x est défini par rapport à y.

Exemple E_i = externe

S_i = section ou module objet.



Les extrémités du graphe sont bien sûrs des sections. On remarque qu'il est nécessaire de résoudre E2 et E3 pour résoudre E1 et finalement E5.
Si G est le graphe initial on calcule tous ses puits, ce sont en effet les seuls éléments calculables à ce stade.
On considère ensuite le graphe G' = G - {puits de G et les arcs menant à ces puits} et on calcule les puits de G', et ainsi de suite jusqu'à épuisement du graphe. Si cette étape ne peut être atteinte, les sommets restants sont les éléments non calculables.

Pratiquement, le fonctionnement du relieur peut se décomposer en trois étapes :

lère étape : constitution de tables de modules objets et d'externes.

2ème étape : le relieur décide de l'ordre d'implantation des sections du programme :
- soit de façon arbitraire (ordre de déclaration des sections),
- soit après une étude de la structure du programme en fonction des informations fournies par le graphe des liens entre qui à pu être constitué.

traduit la définition en terme d'adresse des autres objets.

3ème étape : constitution définitive du code exécutable

C'est bien sûr lors de la seconde phase, après l'étude menée sur la structure du programme que l'on peut agir pour adapter cette structure au milieu physique dans lequel il sera implanté.

La matrice d'affinité statique sera obtenue simplement Elle représente le graphe $G(X,\Gamma)$ définit par :

X : ensemble des blocs

Γ : relation de liaison entre les blocs.

CHAPITRE VI : TEST DES NOUVELLES IMPLANTATIONS.

6.1. Introduction
6.2. Description de notre simulateur

6.1. Introduction.

Disposant de différentes structures d'un programme, nous nous trouvons maintenant devant le problème qui consiste à les comparer.
Différents indices de performance pourraient être utilisés en vue de réaliser ces comparaisons. Parmi ceux-ci, nous avons choisi de nous baser sur le nombre de défauts de page générés lors de l'exécution des programmes.
Cette mesure devant être prélevée dans des conditions identiques pour toutes les structures testées, il ne nous a pas été possible de la réaliser lors d'exécutions réelles des programmes. Il nous aurait en effet été impossible de maîtriser certains paramètres qui l'influencent, tel que par exemple la charge du système. Nous avons donc élaboré un simulateur permettant de placer toutes les structures dans des conditions rigoureusement identiques.

6.2. Description du simulateur.

Le simulateur que nous avons construit utilise une chaîne de références du programme. Comme pour le calcul de l'affinité, nous ne disposons pas ici de la chaîne complète mais d'une sous-chaîne ne comportant

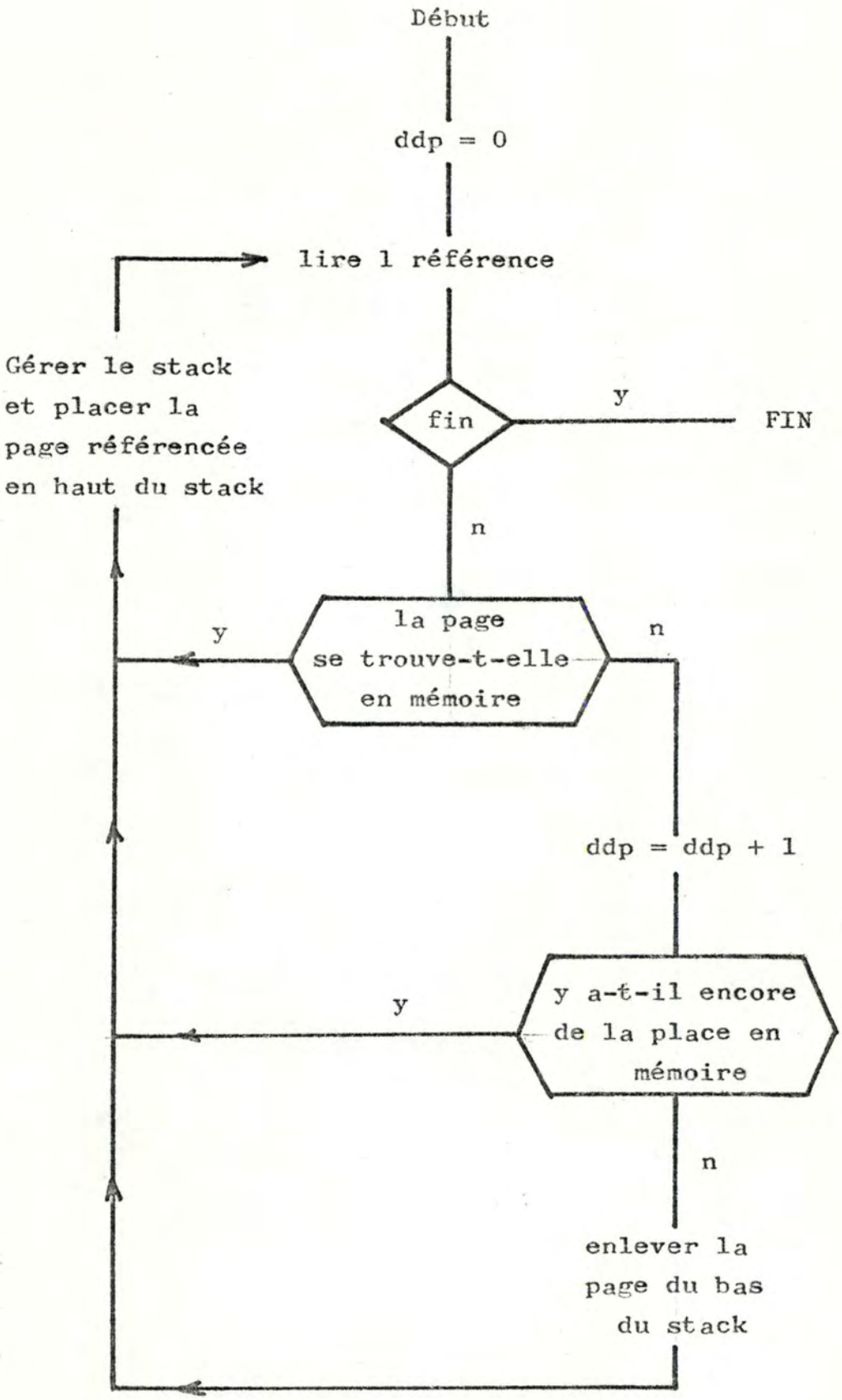
que des passages de contrôles d'un bloc à un autre. La même chaîne sera bien évidemment utilisée lors de chaque simulation.

Le simulateur interprète cette chaîne de référence et simule le mécanisme de chargement des pages dans une mémoire de taille fixe gérée par un algorithme de remplacement de type LRU.

Le comportement simulé est différent du comportement réel, il est toutefois en accord avec les principes généraux de ce dernier et est de plus identique pour chaque structure.

Le fonctionnement de ce simulateur est représenté sur l'algorithme figurant à la page suivante.

La variable ddp contiendra à la fin de la simulation de nombre de défauts de page générés.


Début
ddp = 0
lire 1 référence
Gérer le stack
et placer la
page référencée
en haut du stack
fin
y
FIN
n
la page
se trouve-t-elle
en mémoire
y
n
ddp = ddp + 1
y a-t-il encore
de la place en
mémoire
y
n
enlever la
page du bas
du stack

Exemple :

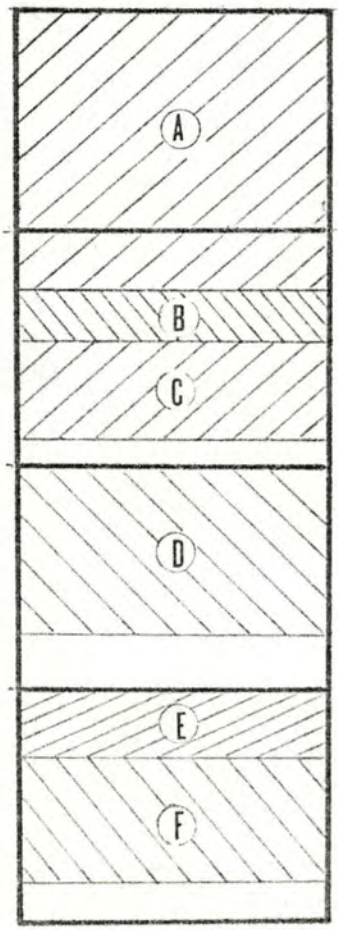
soient :

- la chaîne de référence initiale :

$r_a\ r_b\ r_c\ r_b\ r_d\ r_e\ r_d\ r_b\ r_a\ r_f\ r_a$

- la structure testée :

Bloc	N° de la page dans laquelle il est implanté.
A	1 et 2
B	2
C	2
D	3
E	4
F	4



Page 1.

page 2.

page 3.

page 4.

- taille de la mémoire : 3 cadres.

Simulation :

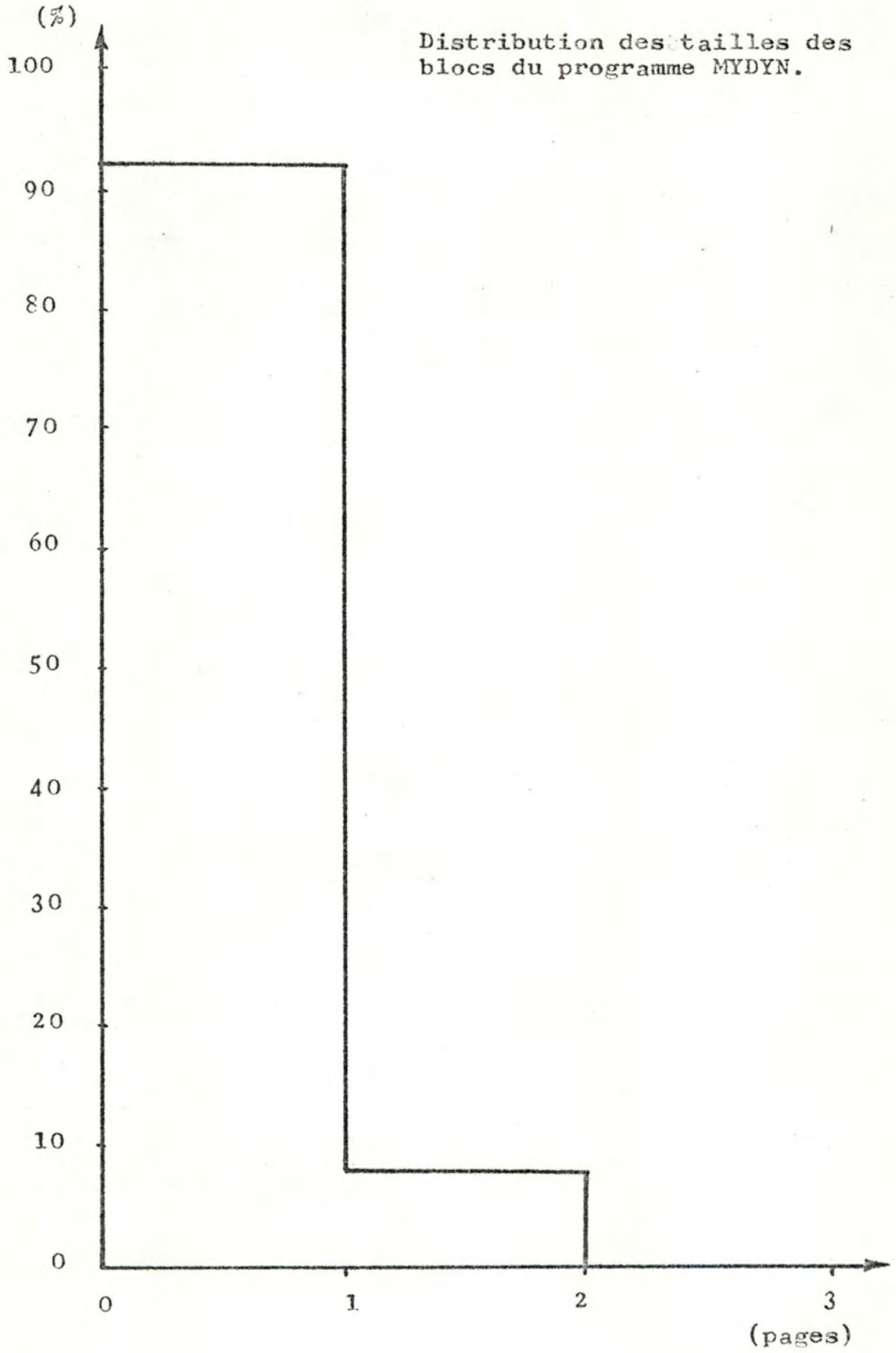
Chaîne initiale	Chaîne intermédiaire	Stack Mémoire	DDP
A	1	1	1
	2	2 1	2
B	2	2 1	
C	2	2 1	
B	2	2 1	
D	3	3 2 1	3
E	4	4 3 2	4
D	3	3 4 2	
B	2	2 3 4	
A	1	1 2 3	5
	2	2 1 3	
F	4	4 2 1	6
A	1	1 4 2	
	2	2 1 4	

CHAPITRE VII : RESULTATS DES TESTS DE QUELQUES METHODES DE CALCUL DE L'AFFINITE ET DE RESTRUCTURATION.

7.1. Introduction
7.2. Résultats

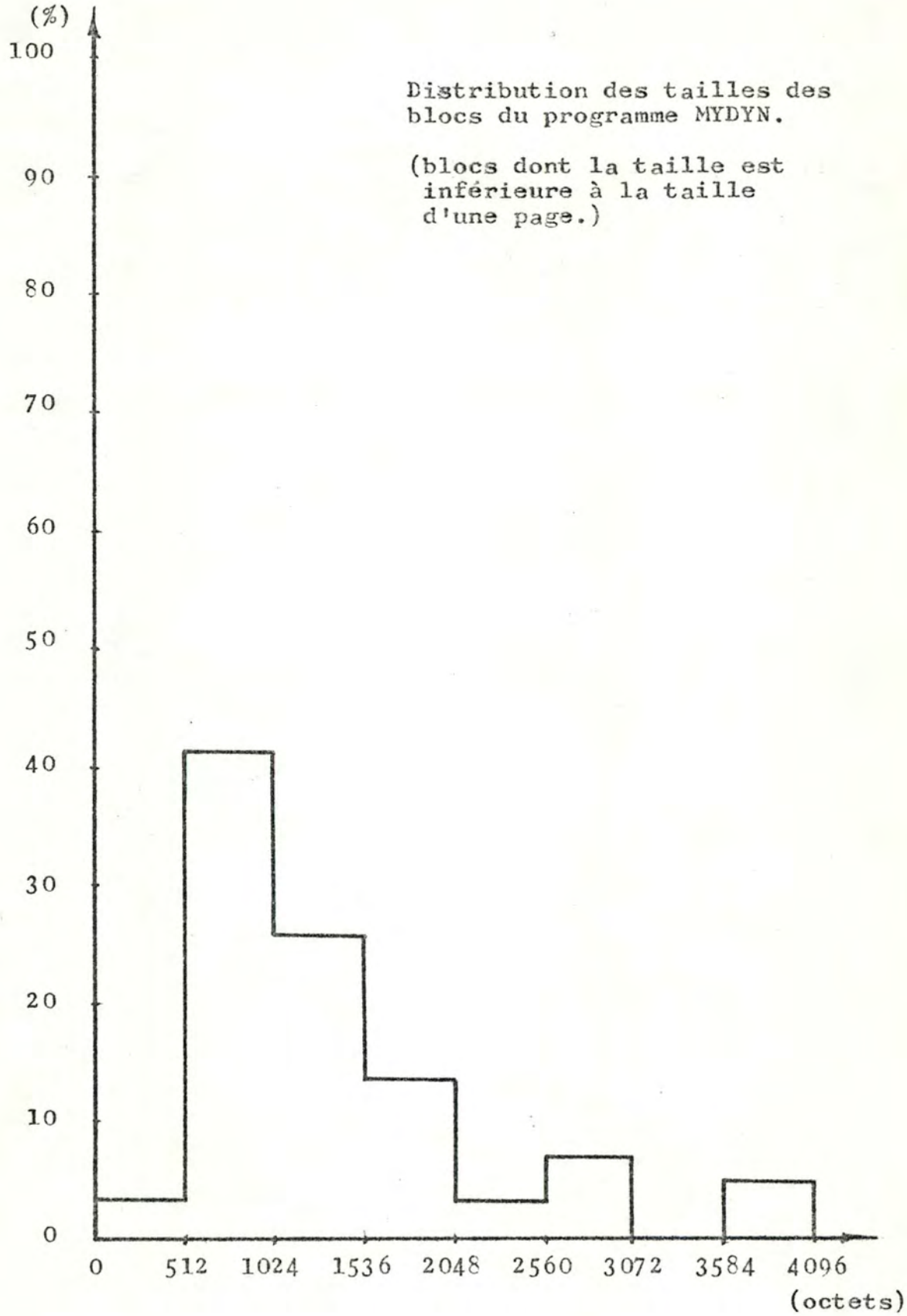
7.1. Introduction.

Le programme que nous avons pu examiner appartient à la famille des pré-compilateurs. Il est constitué de 63 routines FORTRAN que nous assimilerons à des blocs. Sa taille est de 26 pages. La distribution des tailles des blocs est donnée par les figures VII 1. et VII 2. La première correspond à la distribution complète. (On remarquera qu'aucun bloc n'a une taille supérieure à 2 pages), la seconde nous montre la distribution des tailles des blocs pour les blocs dont la taille est inférieure à une page.
La taille moyenne du bloc est de 1680 octets soit 0,41 page. Si l'on ne tient compte dans ce calcul que des blocs dont la taille est inférieure à une page, la taille moyenne du bloc est de l'ordre de 1/3 de page.
Ceci nous place déjà dans des conditions favorables à une restructuration.
La chaîne de références relevée lors d'une exécution de ce programme comportait approximativement 12000 références. Elle correspond à un temps d'exécution du programme de l'ordre de 8 secondes. Tous les blocs n'ont pas été référencés lors de cette exécution. (routines d'erreurs)
Nous avons estimé le nombre de défauts de page générés par ce programme au moyen de simulateur décrit au chapitre précédent.



Distribution des tailles des blocs du programme MYDYN.

Fig. VII 1.



Distribution des tailles des blocs du programme MYDYN.

(blocs dont la taille est inférieure à la taille d'une page.)

Fig VII 2.

7.2. Résultats

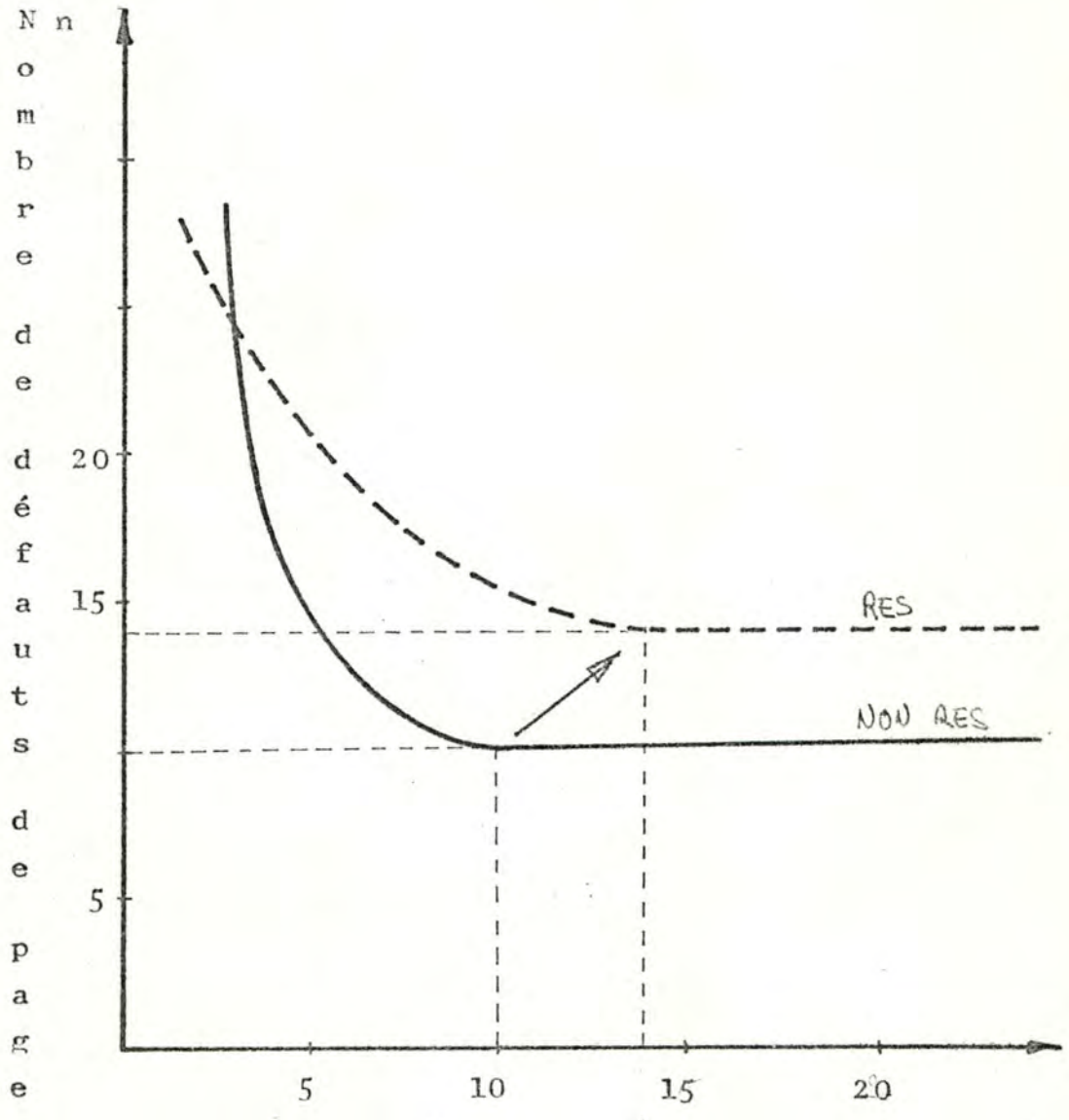
Divers essais de restructuration ont été réalisés au moyen de l'algorithme de RYDER, la matrice d'affinité étant constituée par les méthodes décrites au paragraphe 2.5.3.1. (calcul basé sur l'emplacement des références dans la chaîne). Le seul paramètre modifié d'un essais à l'autre étant la constante C de la fonction $f(n) = k(C - n)$ (fenêtre)

On pourra examiner l'effet de ces restructurations grace aux graphiques qui suivent.
Chacun de ceux-ci correspondant à un choix différent de la taille de la fenêtre utilisée lors du calcul de l'affinité, montre la dépendance du nombre de défauts de page vis à vis de la taille mémoire disponible (en nombre de cadres).

- la courbe en trait plein correspond au nombre de défauts de page générés par le programme initial, c'est à dire structuré sans mesures par le programmeur;
- la courbe en trait interrompu correspond au nombre de défauts de page générés par le programme restructuré.

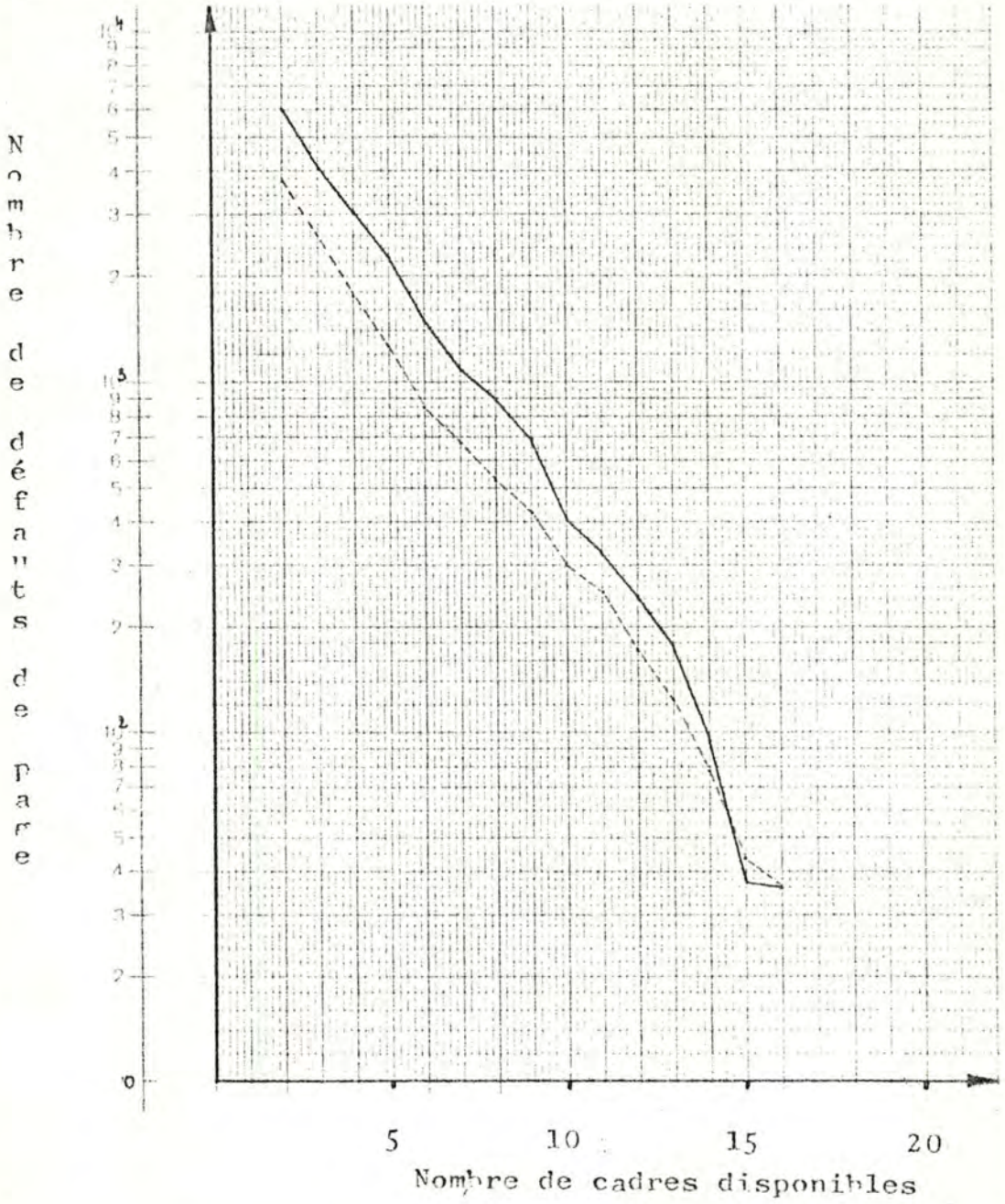
On remarquera que les effets de la restructuration sont les plus marqués pour des fenêtres de taille 5 ou 6 et cela quel que soit le nombre de cadres alloués au programme.
Quand le nombre de cadres alloués au programme est proche ou supérieur à la taille de ce dernier, on remarque que le nombre de défauts de page générés par le programme restructuré est plus élevé que le nombre de défauts de page générés par le programme initial.
Ce phénomène est dû au fait que les programmes restructurés par des algorithmes tenant compte de la pagination de la mémoire occupent plus de pages après restructuration qu'avant.

Les courbes représentant le nombre de défauts de page en fonction de la taille mémoire disponible présentent toujours une asymptote horizontale.
En effet, quand le nombre de cadres disponibles est égal ou supérieur au nombre de pages du programme, le nombre de défauts de page est égal à ce dernier nombre (aucune page n'étant remplacée.) Il est évident que l'asymptote correspondant au programme restructuré est toujours à un niveau plus élevé que celle correspondant au programme non restructuré.



TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



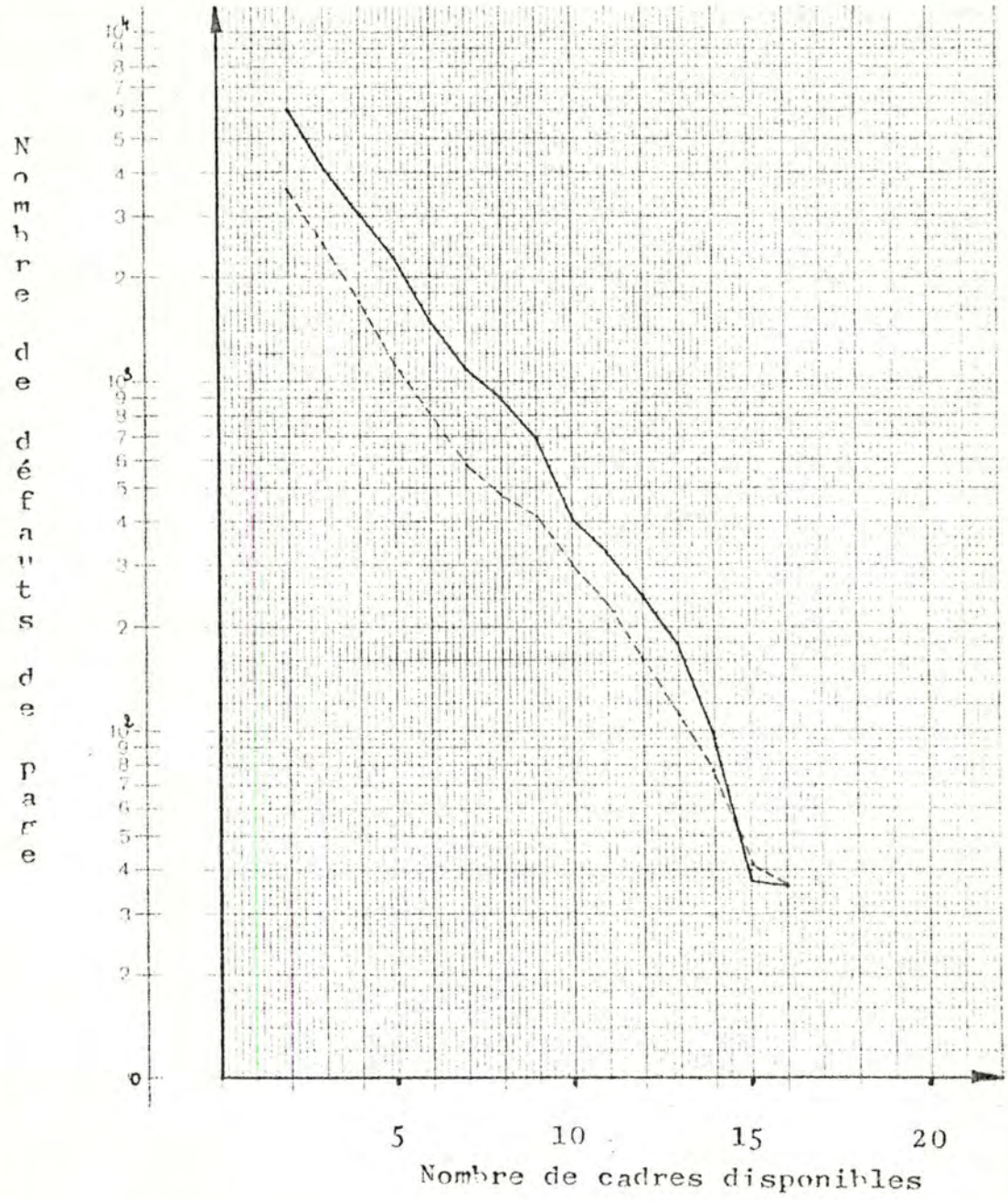
LEGENDE :

_____ : programme non restructuré.

----- : programme restructuré.
affinités calculées au moyen d'une fenêtre de taille 2.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



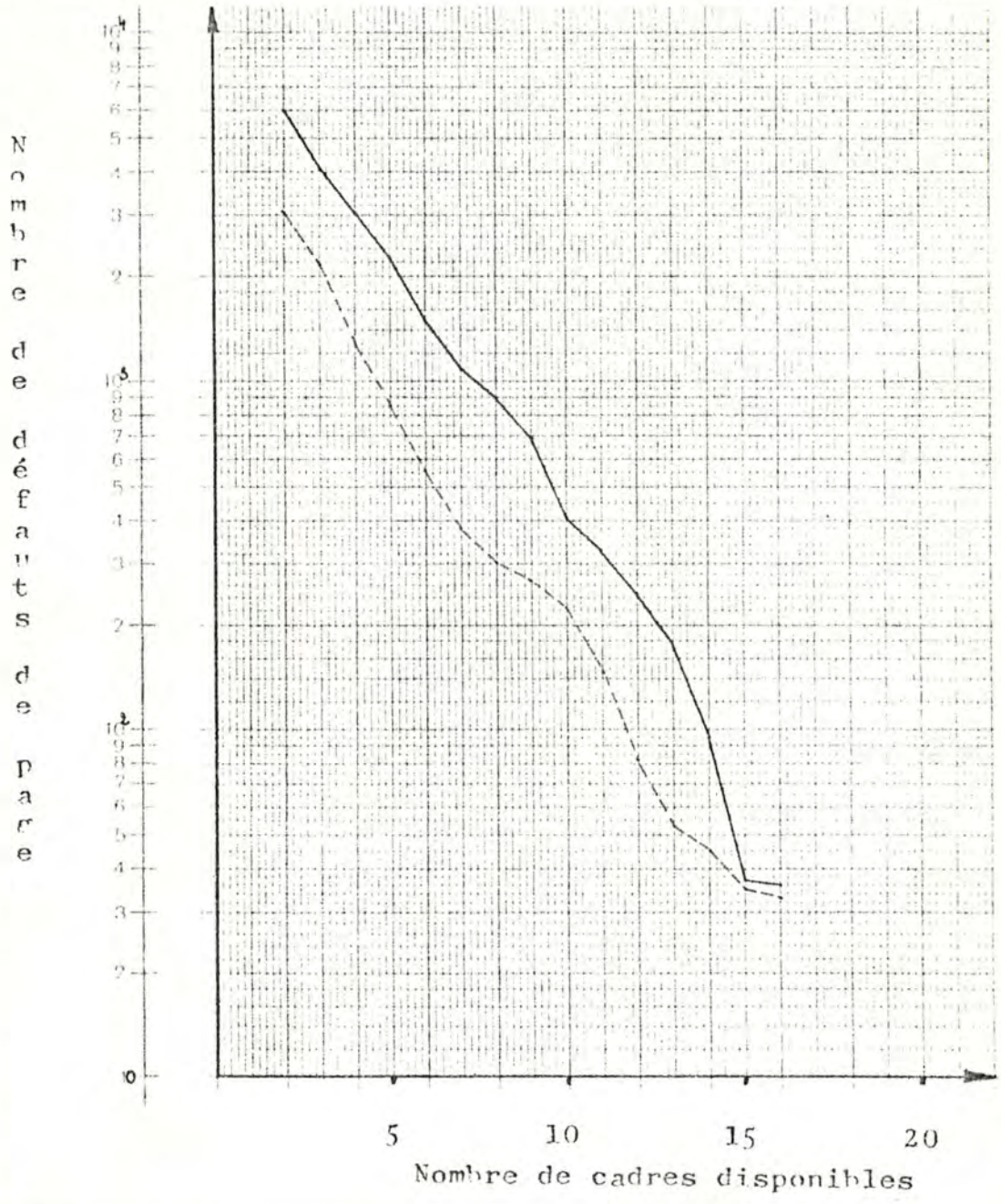
LEGENDE :

——— : programme non restructuré;

----- : programme restructuré. affinités calculées au moyen d'une fenêtre de taille 3.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



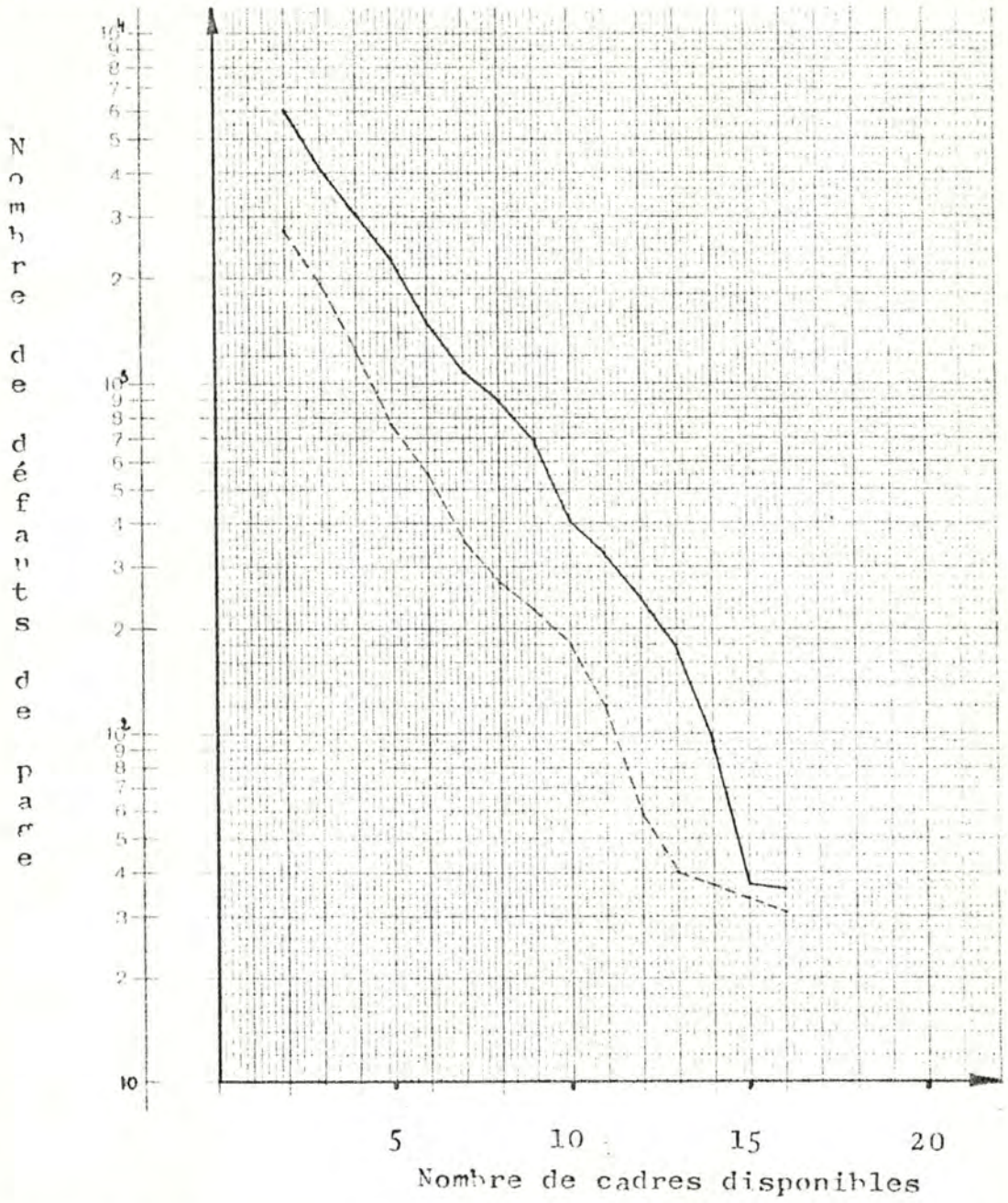
LEGENDE :

——— : programme non restructuré.

----- : programme restructuré. affinités calculées au moyen d'une fenêtre de taille 4.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



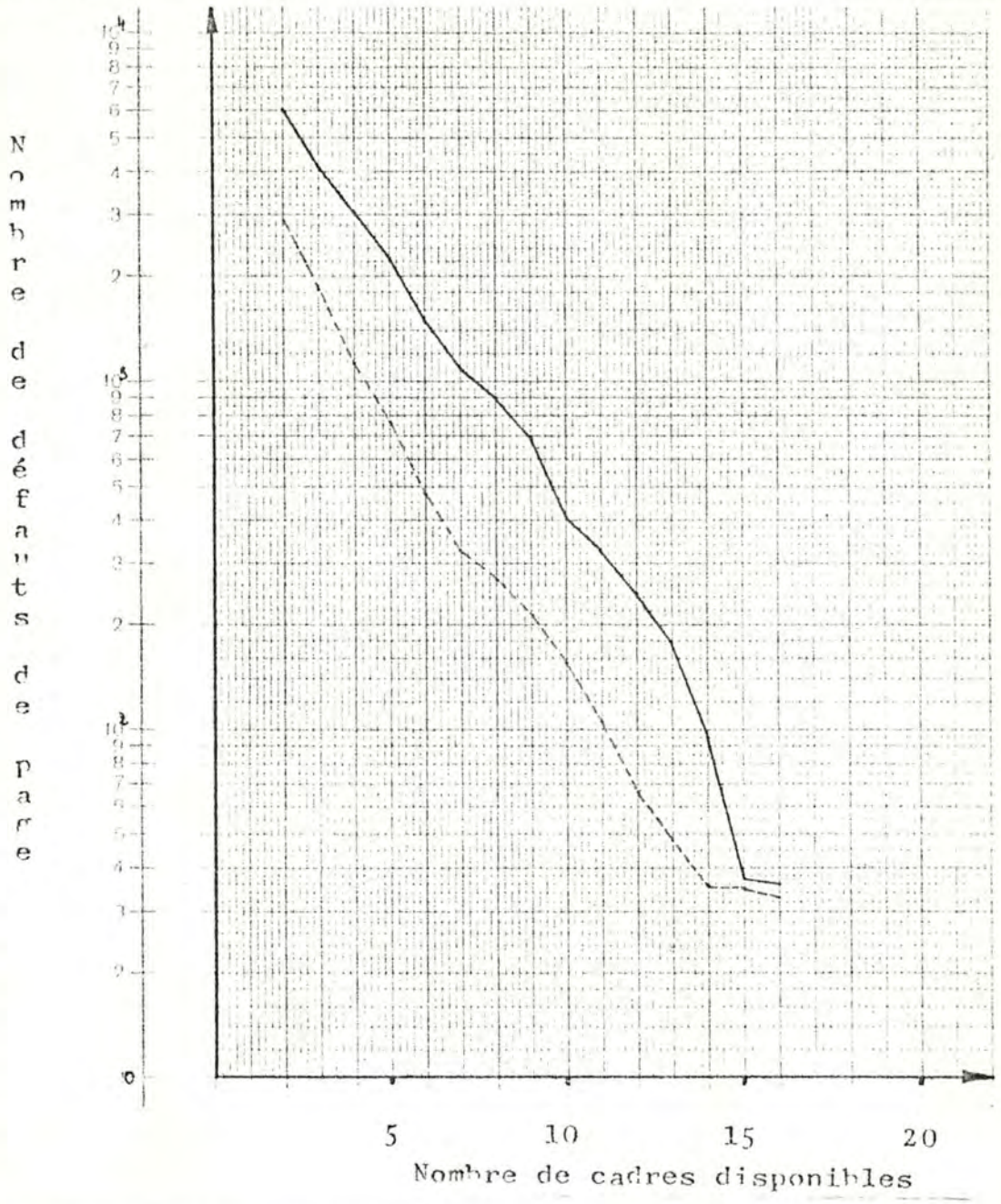
LEGENDE :

——— : programme non restructuré.

----- : programme restructuré. affinités calculées au moyen d'une fenêtre de taille 5.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



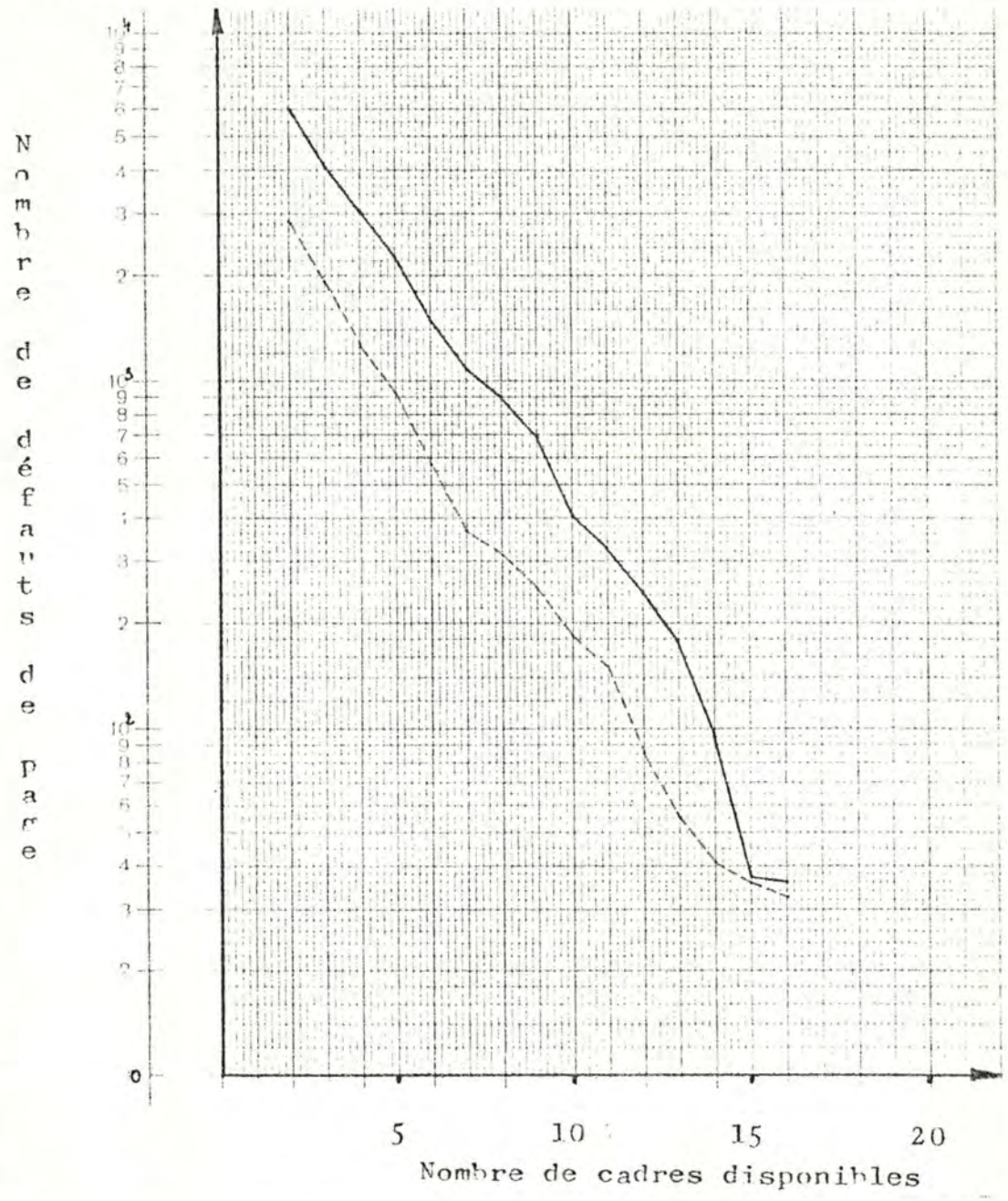
LEGENDE :

——— : programme non restructuré.

----- : programme restructuré. affinités calculées au moyen d'une fenêtre de taille 6.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



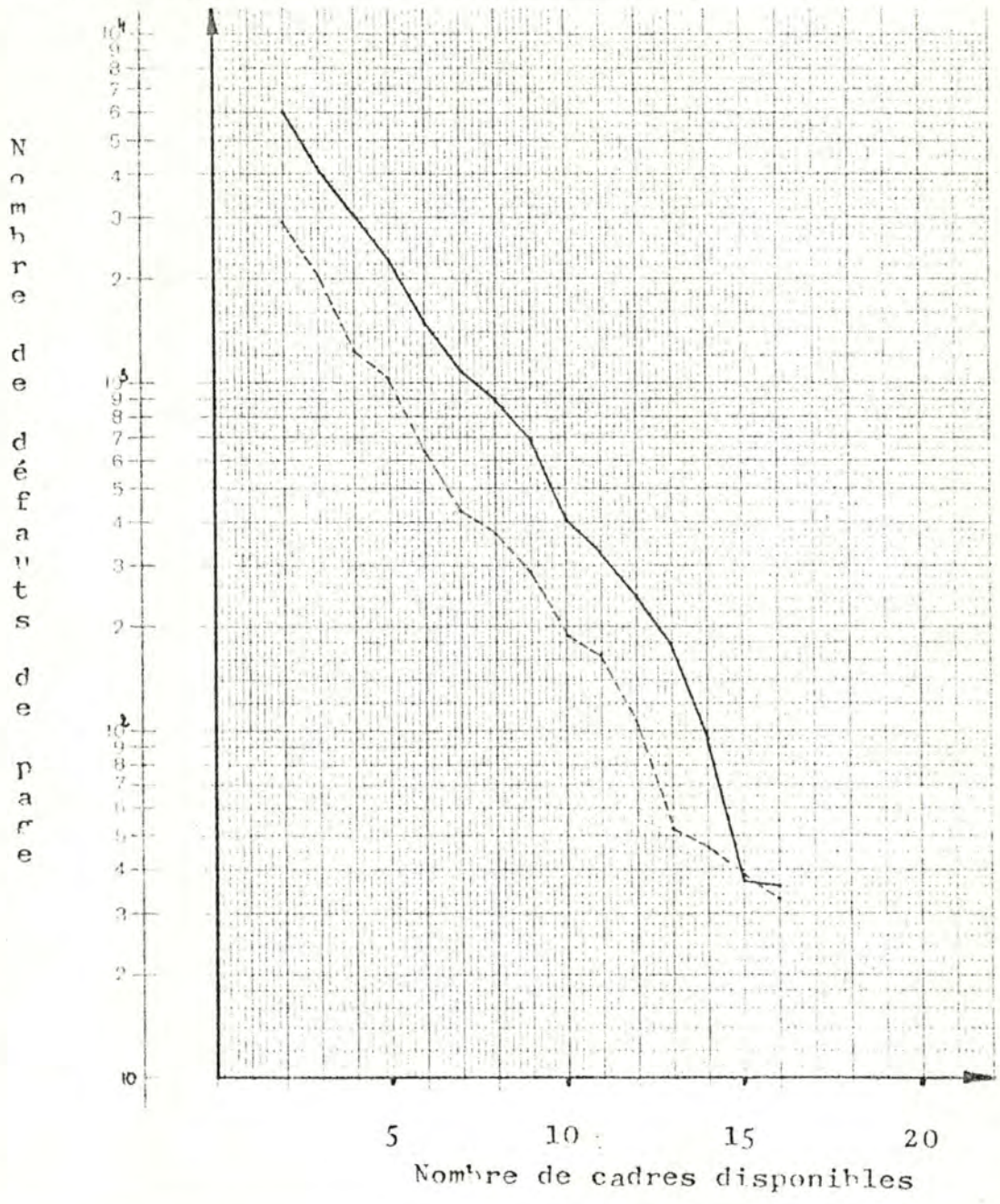
LEGENDE :

——— : programme non restructuré.

----- : programme restructuré. affinités calculées au moyen d'une fenêtre de taille 7.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page

PROGRAMME : MYDYN



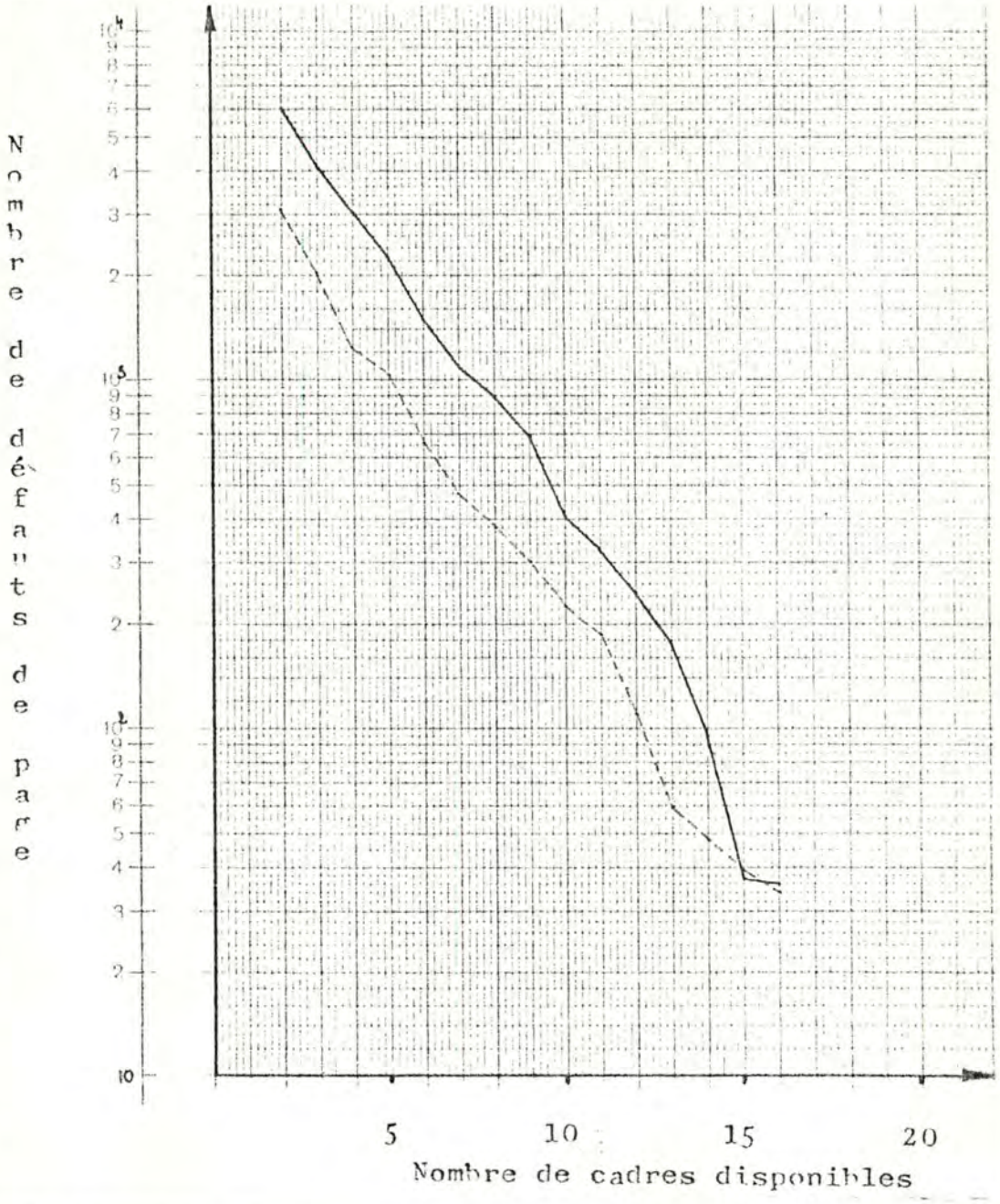
LEGENDE :

——— : programme non restructuré.

----- : programme restructuré.
affinités calculées au moyen d'une fenêtre de taille 8.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



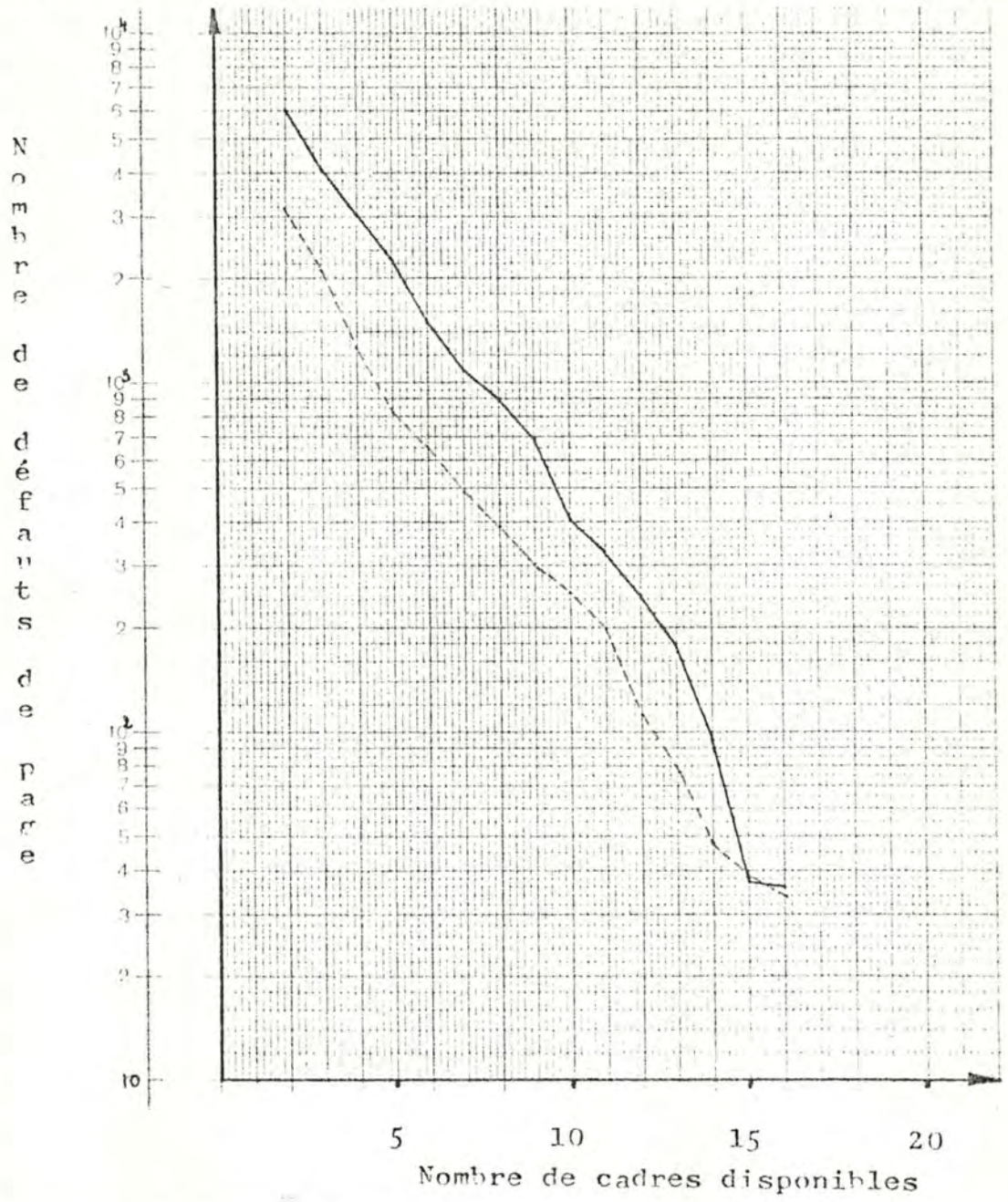
LEGENDE :

——— : programme non restructuré.

----- : programme restructuré.
affinités calculées au moyen d'une fenêtre de taille 9.

TITRE : Influence d'une restructuration par l'algorithme de RYDER sur le nombre de défauts de page.

PROGRAMME : MYDYN



LEGENDE :

——— : programme non restructuré.

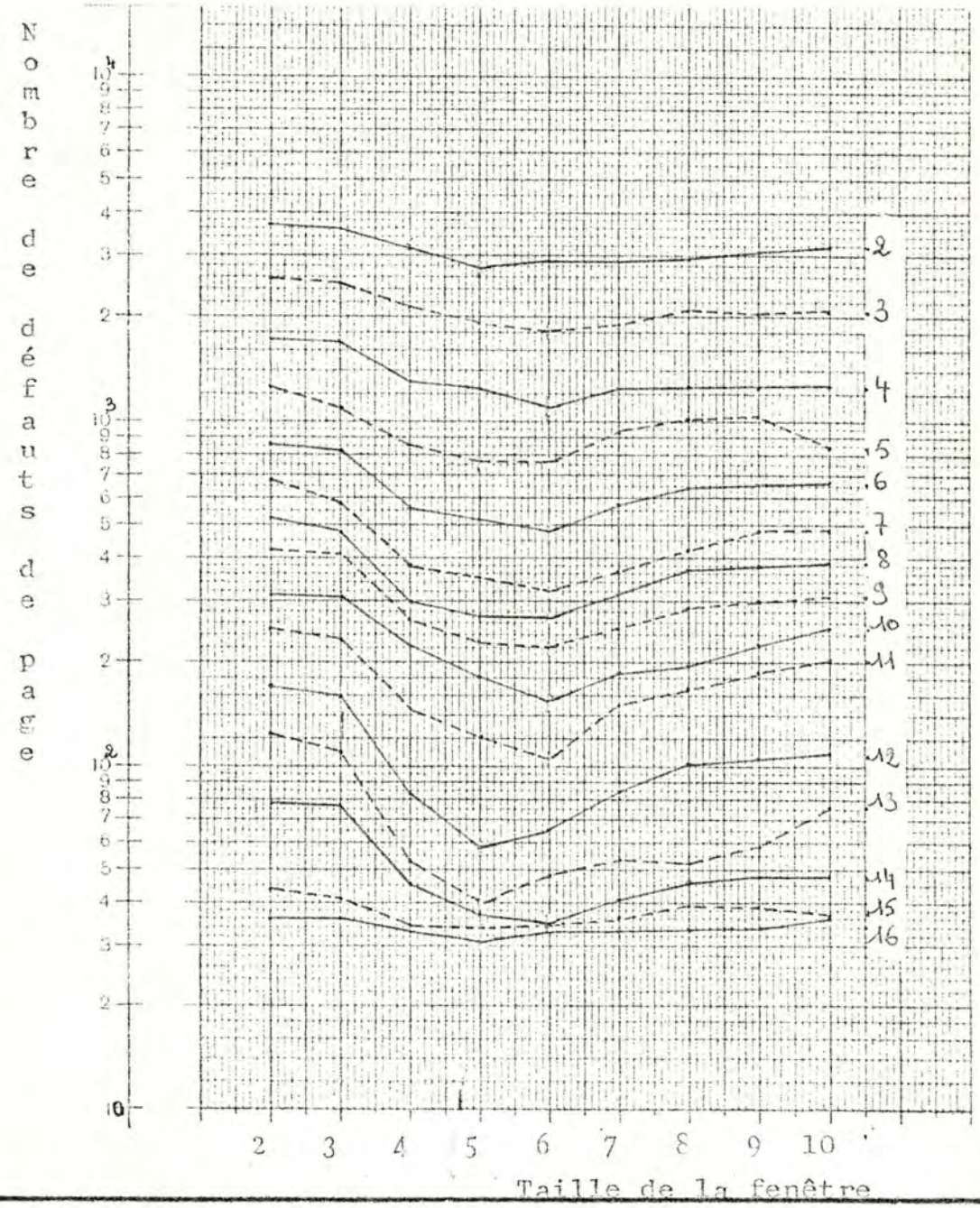
----- : programme restructuré. affinités calculées au moyen d'une fenêtre de taille 10.

L'effet du choix de la taille de la fenêtre utilisée pour le calcul de l'affinité est plus clairement visible sur le graphique qui suit.
Chacune des courbes qui y figure montre la dépendance du nombre de défauts de page vis à vis de la taille de la fenêtre utilisée pour le calcul de l'affinité et ce pour une disponibilité en mémoire fixée.
Chacune de ces courbes présente un minimum pour une fenêtre de taille 5 ou 6.

TITRE : RESTRUCTURATION PAR L'ALGORITHME DE RYDER.

Influence de la taille de la fenêtre utilisée lors du calcul des affinités sur le nombre de défauts de page.

PROGRAMME : MYDYN



LEGENDE :

Chacune des courbes correspond à la disponibilité en cadres mémoire indiquée à sa droite.

Densification

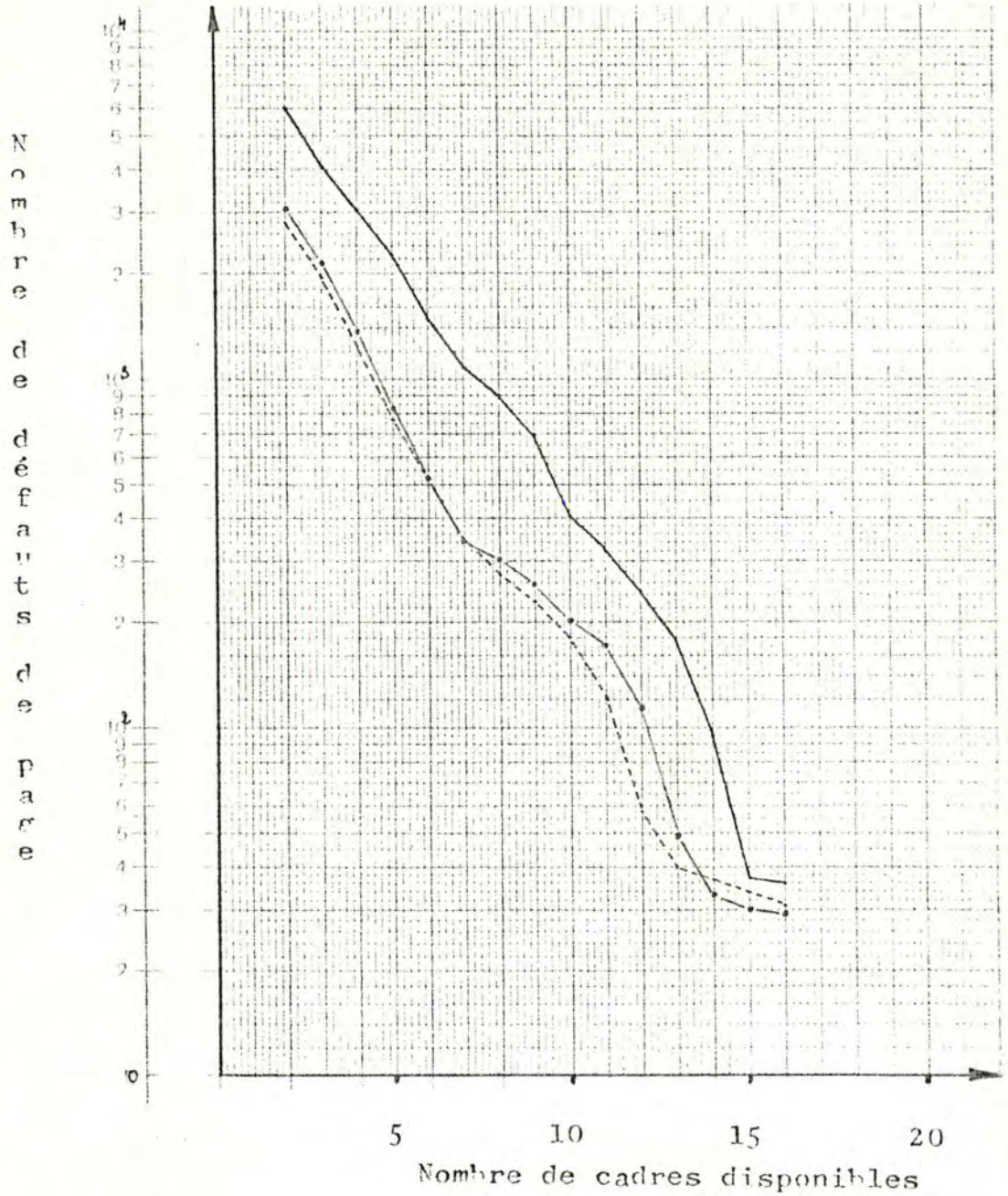
L'influence de la restructuration étant défavorable dans certains cas du fait de la taille accrue du programme restructuré, il nous a paru utile de rendre au programme restructuré sa taille initiale en le densifiant.

Les deux graphiques qui suivent montre le résultat d'une telle densification de 2 structures obtenues précédemment (calcul de l'affinité par fenêtre de taille 5 et 10, algorithme de RYDER)

L'influence de cette densification est défavorable quand l'espace mémoire disponible est peu important. Elle corrige toutefois l'effet de l'accroissement de la taille du programme décrit précédemment.

TITRE : Influence de la densification des programmes restructurés sur le nombre de défauts de page.

PROGRAMME : MYDYN



LEGENDE :

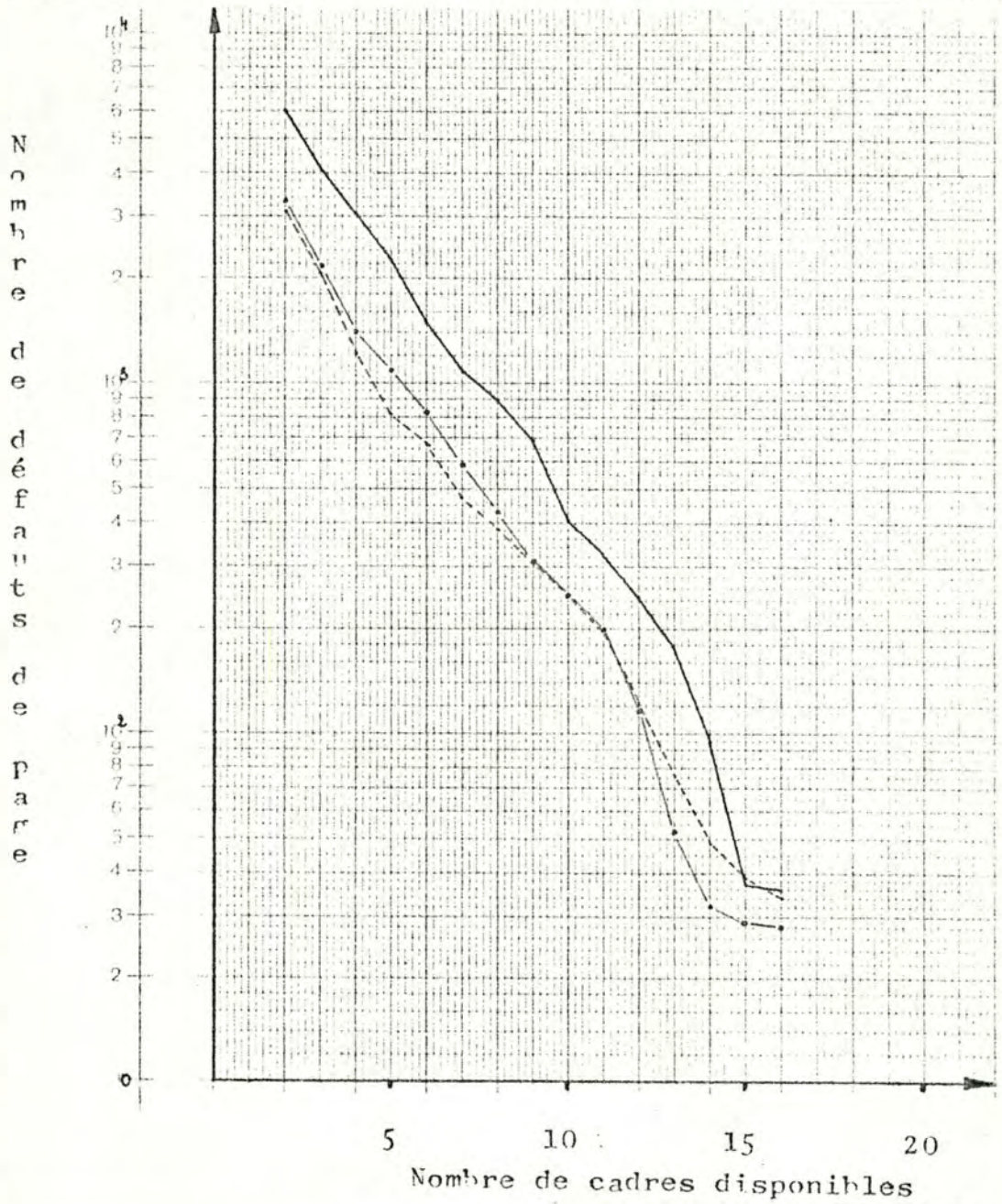
——— : programme non restructuré.

----- : programme restructuré. Fenêtre 5.

—.— : programme restructuré et densifié.

TITRE : Influence de la densification des programmes restructurés sur le nombre de défauts de page

PROGRAMME : MYDYN



LEGENDE :

——— : programme non restructuré.

----- : programme restructuré. Fenêtre 10.

—.— : programme restructuré et densifié.

Deux essais de restructuration ont été réalisés à partir de méthodes d'essaimage.
En premier lieu, une restructuration par l'algorithme de MASUDA basée sur une matrice d'affinité constituée au moyen de la fonction f(n) = 4 - n (fenêtre de taille 5) ensuite une restructuration tenant compte de la mise à jour de la matrice par la méthode de "l'affinité la plus élevée" à partir de la même matrice d'affinité.

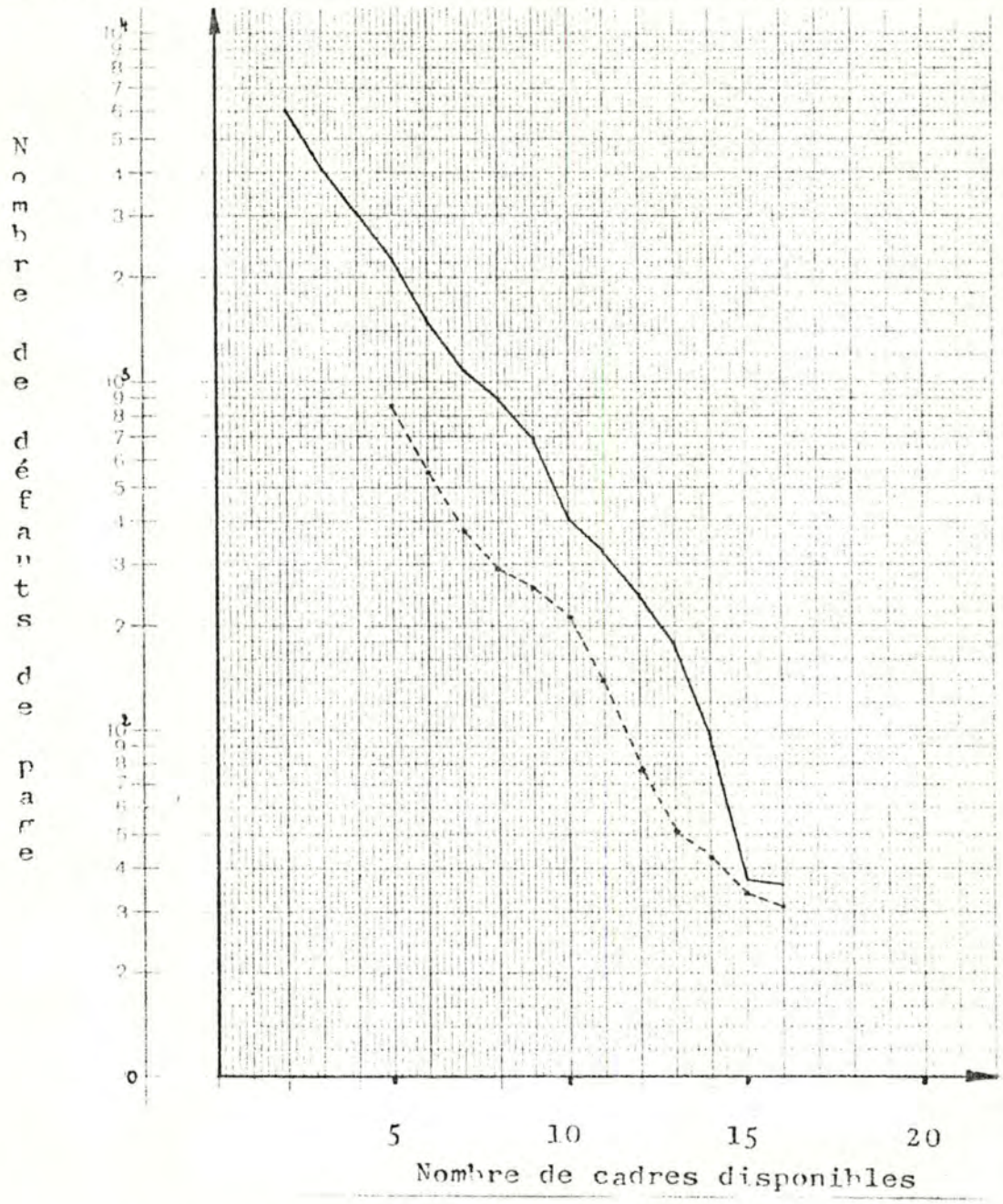
Les courbes obtenues sont toutes deux caractéristiques des classes des algorithmes de restructuration.

1) l'algorithme de MASUDA présente le même défaut que l'algorithme de RYDER, (inefficacité quand un espace disponible en mémoire a une taille proche ou supérieure à celle du programme)

2) par contre, l'autre méthode présente une efficacité élevée et constante qui nous à d'ailleur étonné.
   En effet, quand nous avons utilisé cet algorithme en vue de restructurer les 63 blocs du programme MYDYN, il a produit une structure ne comportant que 25 blocs (les autres algorithmes produisant toujours une structure quasi-complète)
   Les 38 blocs restants ont été adjoints à cette structure dans un ordre quelconque.

TITRE : Influence d'une restructuration par l'algorithme de MASUDA sur le nombre de défauts de page

PROGRAMME : MYDYN



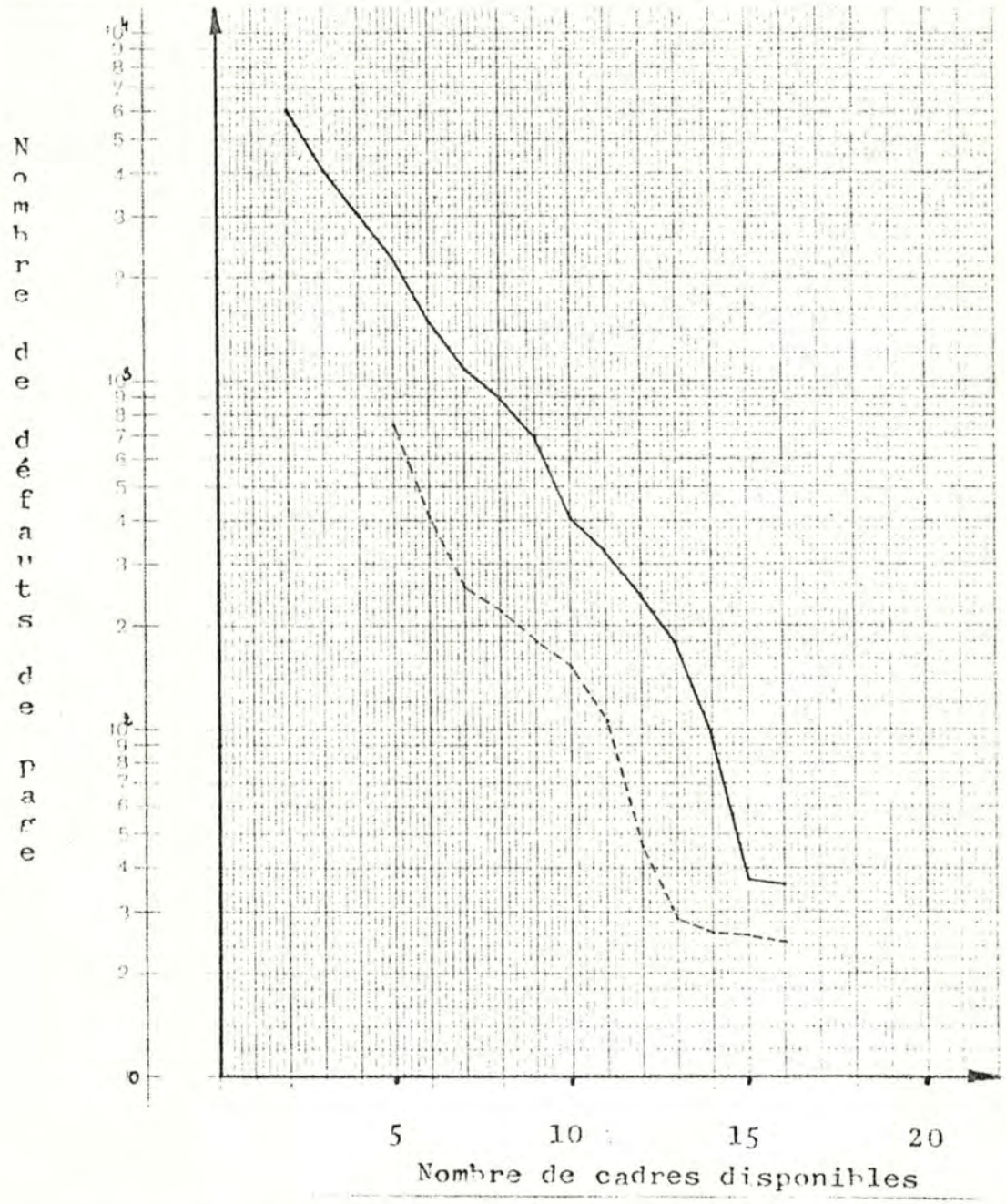
LEGENDE :

——— : programme non restructuré

----- : programme restructuré fenêtre 5.

TITRE : Influence d'une restructuration par essaimage sur le nombre de défauts de page. (affinité la plus élevée.)

PROGRAMME : MYDYN



LEGENDE :

——— : programme non restructuré.

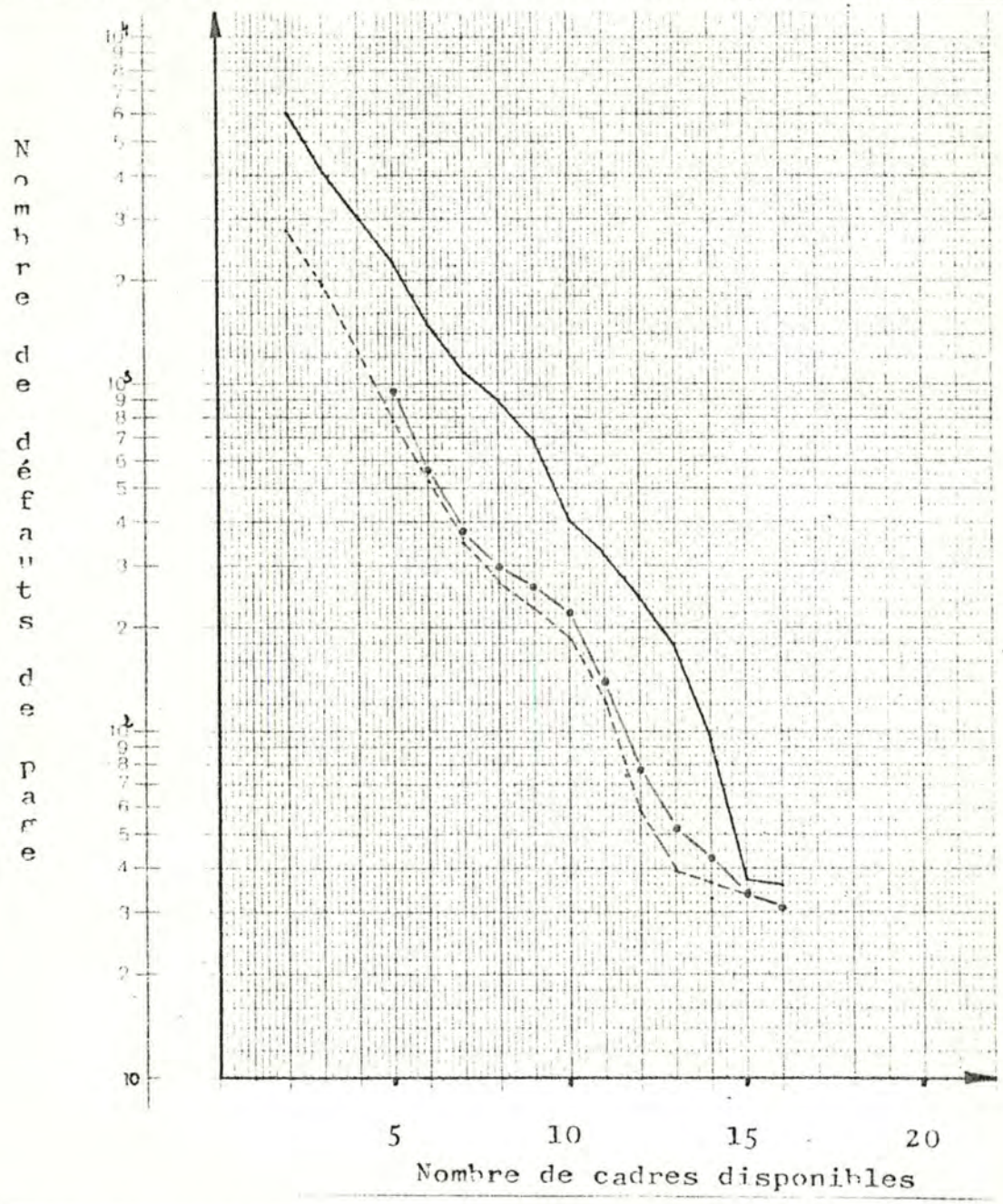
----- : programme restructuré affinités calculées au moyen d'une fenêtre de taille 5.

Nous avons comparé des résultats avec ceux obtenus par l'algorithme de RYDER appliqué à la même matrice d'affinité.

On remarque que l'algorithme de MASUDA fournit constemment de moins bons résultats, tandis que l'autre méthode donne une structure plus performante que celle obtenue par l'algorithme de RYDER densifiée (ou non.)

TITRE : Comparaison des algorithmes de RYDER et de MASUDA.

PROGRAMME : MYDYN



LEGENDE :

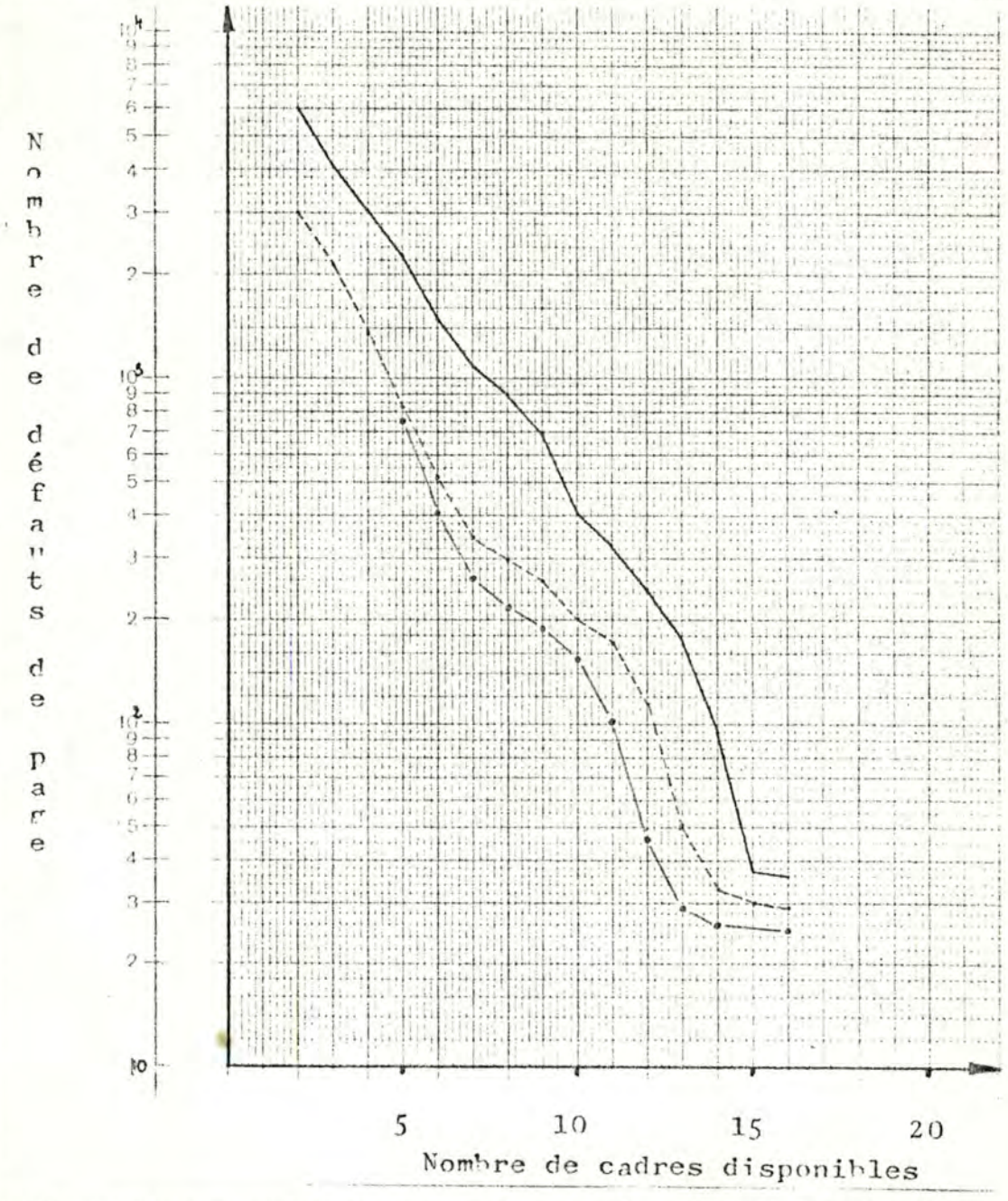
_____ : programme non restructuré.

—.—. : programme restructuré par l'algorithme de MASUDA.

---- : programme restructuré par l'algorithme de RYDER.

TITRE : Comparaison de l'algorithme de RYDER et de la restructuration par essaimage. (affinité la plus élévée).

PROGRAMME : MYDYN



LEGENDE :

——— : programme non restructuré.

—·— · : programme restructuré par essaimage.

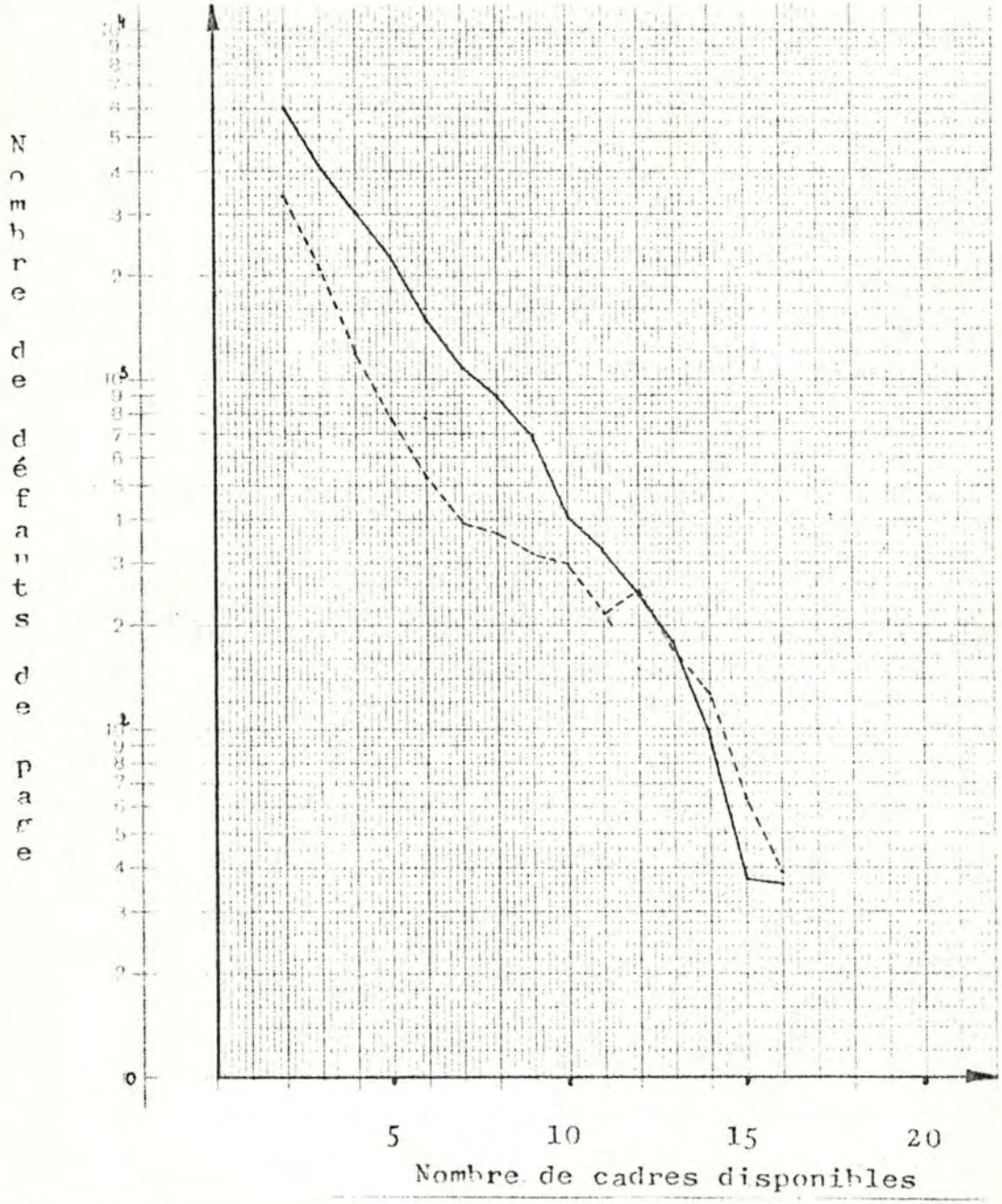
---- : programme restructuré par l'algorithme de RYDER.

Nous avons finalement testé les différentes méthodes de calcul de l'affinité basées sur des échantillons de chaîne de référence.
Celle-ci a été échantillonnée par rapport au temps la première fois en considérant des échantillons de 5 ms pris toutes les 50 ms et ensuite des échantillons de 10 ms considérés toutes les 100 ms.
L'affinité a ensuite été calculée par chacune des deux méthodes exposée au paragraphe 2.5.3.3.
La nouvelle structure a été obtenue par l'algorithme de RYDER.
Le nombre de défauts de page ne semble pas être fortement influencé par ces méthodes de calcul de l'affinité. L'échantillonnage influence par cette méthode la performence de la nouvelle structure. Il apparait ainsi que les échantillons de 10 ms sont mal adaptés à notre problème.

Les graphiques qui suivent correspondent à une restructuration réalisée à partir d'une matrice d'affinité obtenue par la méthode multiplicative.

TITRE : Influence de l'échantillonnage de la chaîne de référence.

PROGRAMME : MYDYN



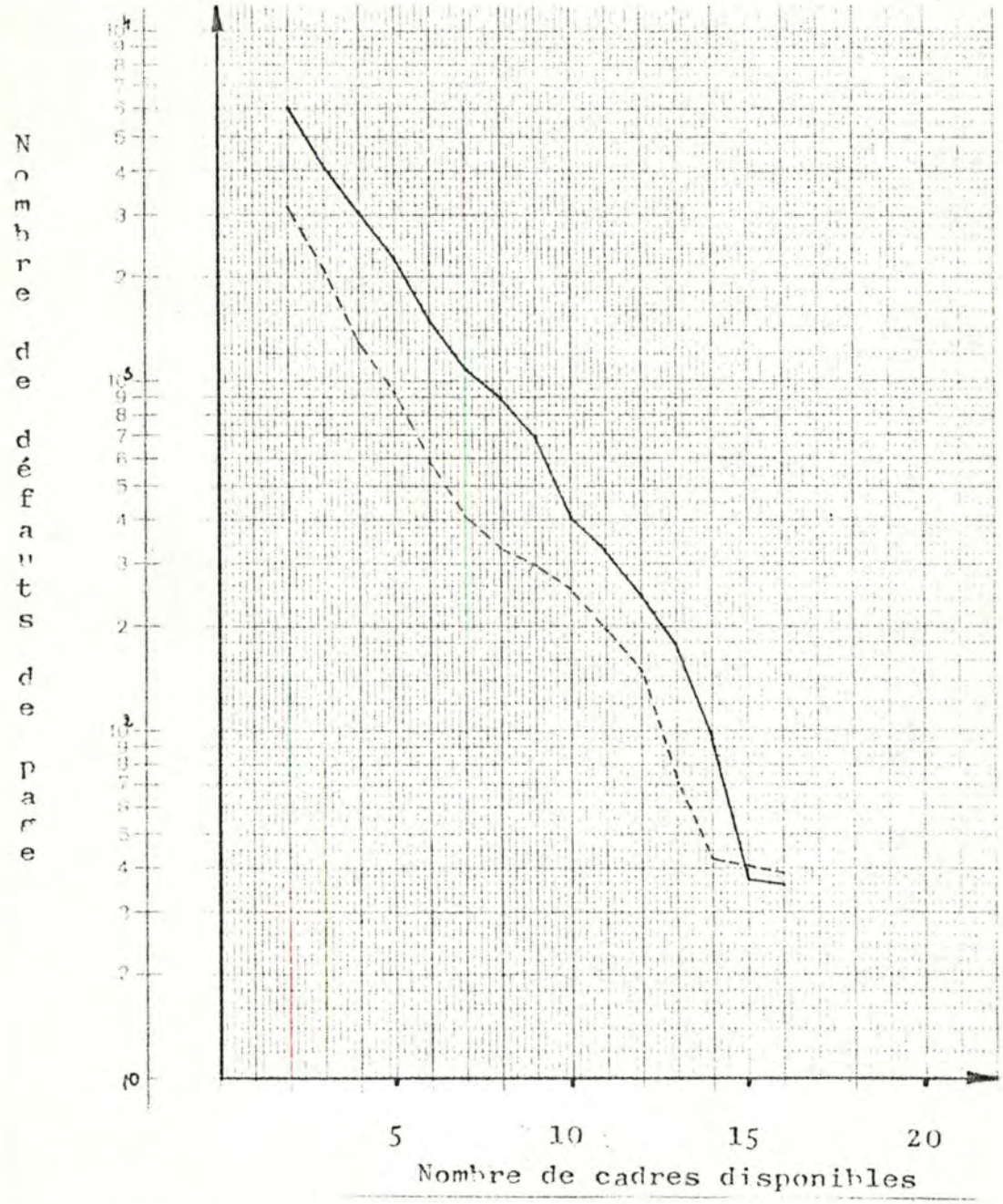
LEGENDE :

——— : programme non restructuré.

---- : programme restructuré (échantillons de 10 ms pris toutes les 100 ms)

TITRE : Influence de l'échantillonnage de la chaîne de références.

PROGRAMME : MYDYN



LEGENDE :

——— : programme non restructuré.

---- : programme restructuré. (échantillons de 5 ms pris toutes les 50 ms).

CHAPITRE VIII : CONCLUSIONS

Nous nous proposions dès le début de cette étude de restructurer des programmes en vue de les adapter au milieu paginé, et donc de diminuer leurs taux de défauts de page.

L'exemple que nous avons considéré (le programme MYDYN) nous a donné entièrement satisfaction, les améliorations obtenues dépassent nettement les prévisions que nous avions pu faire à partir des résultats proposés par les auteurs des différents algorithmes de calcul de l'affinité et de restructuration. Ce fait provient sans doute d'une bonne adaptation du programme MYDYN à la restructuration (taille des blocs).

Dans le cadre des expériences que nous avons réalisées deux paramètres peuvent influencer le nombre de défauts de page générés par le programme étudié lors de son exécution (simulée) dans un espace mémoire de taille fixe :

1) le calcul de l'affinité :
   a) nous avons constaté l'existence d'une fenêtre de taille optimale lors du calcul de l'affinité par la méthode exposée au paragraphe 2.5.3.1 au moyen de la fonction $f(n) = k(C - n)$. (si la restructuration ultérieure est réalisée par l'algorithme de RYDER).
   b) on remarquera qu'il convient d'être particulièrement prudent lors du calcul de l'affinité par échantillonnage de la chaîne de références. (influence de la taille des échantillons).

2) les algorithmes de restructuration :
   a) bien que l'ensemble des algorithmes de restructuration étudiés nous ait donné satisfaction du point de vue de la réduction du nombre de

défauts de page, il nous semble que les algorithmes basés sur les techniques d'essaimage soient particulièrement intéressants du fait de :

- leur simplicité de programmation,
- leur vitesse d'exécution (coût de la nouvelle structure).
- la qualité des structures qu'ils produisent.

b) Nous avons remarqué que les algorithmes indépendants de la pagination de la mémoire produisent des structures plus efficaces que celles produites par les algorithmes dépendants de la pagination quand le nombre de cadres mémoire est égal ou supérieur au nombre de pages du programme.

Il pourrait nous être reproché de ne pas avoir étudié la restructuration d'autres programmes, ce qui nous aurait sans doute permis de tirer des conclusions plus générales. On notera toutefois que dans le cadre de la réalisation d'un mémoire, forcément limitée, il n'est pas toujours possible de réaliser toutes les expériences souhaitées.

B I B L I O G R A P H I E

/01/ ACHARD — Segmentation automatique des programmes indépendamment des langages de programmation.
Thèse présentée à l'Université de Paris VI
(1975)

/02/ ANDERBERG — Cluster analysis for application
Academic Press Inc.
(1973)

/03/ BARD — Performance criteria and measurement for a time-sharing system
IBM Syst. J. 10, 193
(1971)

/04/ BARD — Characterization of programm paging in a time-sharing environment.
IBM J. of R. & D. 17,387
(1973)

/05/ BELADY — A study of replacement algorithms for a virtual storage computer.
IBM Syst. J. 5,78
(1966)

/06/ BETOURNE — Mesures sur un système conversationnel
IRIA : rapport de recherche 46
(1974)

/07/ BRAWN — Gustavson
Program behavior in a paging environment
Proc. APIPS, Fall joint Computer Conf.
Vol 33, pages 1019-1032
(1968)

/08/ BROWN — Optimum packing and depletion
Mac Donald
American Elsevier Computer Monographs

/09/ BRYANT — Predicting working-set sizes.
IBM J. of R. & D. 19,221
(1975)

/10/ COFFMAN — Varian
Futher experimental data on the behavior of program in a paging environment.
Comm. ACM, 11,7 pages 471-474,
(1968)

/11/ COFFMAN RYAN
A study of storage pratitioning using a mathematical model of locality.
Comm. ACM 15, 185
(1972)

/12/ COMEAU
A study of the effect of user program optimization in a paging system.
ACM symp. on operating systems principles,
Gatlinbourg, Tennessee,
(1967)

/13/ DENNING
The working-set model for program behavior
Comm. ACM, 11, 5, pages 323-333
(1968)

/14/ DENNING
Trashing : its causes end prevention.
Proc. AFIPS 1968, Fall Joint Computer Conf. Vol 33, pages 915-922

/15/ DENNING
Virtual memory.
Computing Surveys 2, n°3 153-189
(1970)

/16/ DENNING
Properties of the working-set model
Com. ACM 15, 3
(1972)

/17/ FERRARI
A toll for automatic program restructuring.
ACM National Conf. Atlanta, Georgia
228-231
(1973)

/18/ FERRARI
Improving locality by critical working-sets
Comm. ACM 17, 614
(1974)

/19/ FERRARI
Improving programmlocality by strategy-oriented restructuring.
Information precessing 1974
North Holland publishing Company.

/20/ HATFIELD GERALD
Program restructuring for virtual memory
IBM Syst. J. N° 3. 19
(1971)

/21/ MADISON BATSON
Characteristics of program localities.
Comm ACM 19, 5, 285
(1976)

/22/ MASUDA — SHIOTA NOGUCHI OHKI
Optimisation of program organisation by cluster analysis.
(1974)

/23/ MATTSON — GECSEI SLUTZ TRAIGER
Evaluation techniques for storage h hierarchies.
IBM Syst. J. 9. 2. 78
(1970)

/24/ MORRISON — User program performance in virtnal storage system.
IBM Syst. J. 3. 216
(1973)

/25/ MORISSET — Adaptation automatique des programmes au milieu paginé.
Thèse présentée à l'Université de Paris.
(1975)

/26/ ROGERS — Structured programming for virtual storage systems.
IBM Syst. J. 4. 385
(1975)

/27/ RYDER — Optimizing program placement in virtual systems.

/28/ SHEDLER — Locality in page reference strings.
SIAM J. on Comp. 1, 3, 218
(1972)